EXPRESS MAIL NO. EV335524655US

TERA MTA Principles of Operation

TERA Computer Company
2815 Eastlake Ave East
Seattle, WA 98102

November 18, 1997
(Composite Revision: 4.263)

Preface

This document is constantly evolving.

[[Details that were being rethought when the document was printed appear in this type style; depending on the context. such a note indicates that additional explanatory text is needed. the design has not been thought out. or the feature being described is deprecated.]]

Contents

List of Figures

Structures

Enumerations

Chapter 1: Introduction

1.1 Notation

This document defines structure and enumeration data types for use by system software. An enumeration definition names the enumeration and the members, gives the integral value of each member, and may give one or more columns of commentary.

A structure definition names the structure (typically of a hardware register) and describes the fields. Fields are written using the notation "Bits *hbn–lbn*" where *hbn* is the high bit number and *lbn* is the low bit number. The width of the field is $hbn - lbn + 1$. Each field has a field name, a type, and one or more columns of commentary text. The field type is either a predefined type or an enumeration type declared elsewhere.

The enumeration names, enumeration members, structure names, field names, and base types all appear in the index. The enumerations and structures defined in this manual are available for use in assembly language and C programs, including the assembler and compilers themselves.

The notation used for operations and instructions is described in §11.1.

Some descriptions in this document include program fragments. Fragments are formatted so that keywords appear in bold face and comments appear in italics.

1.2 Data Types

The memory system can load and store eight-bit bytes, 16-bit quarterwords (2 bytes), 32-bit halfwords (4 bytes), or 64-bit words (8 bytes). Bits are numbered from right to left: the least significant bit is bit number 0.

The most important architecturally supported data types are these:

bit vector
A bit vector may be of any length and may span one or more word boundaries.

signed integer
Signed integers are interpreted in two's complement. Byte, quarterword, and halfword signed integers are sign-extended to 64 bits when they are loaded and quietly truncated to the proper length when they are stored.

unsigned integer
Byte, quarterword, and halfword unsigned integers are zero-extended to 64 bits when they are loaded and quietly truncated to the proper length when they are stored.

floating point
Floating-point numbers and operations conform to IEEE Standard 754. Single (32-bit) and double (64-bit) basic formats are supported. Support for a 128-bit floating-point format is also provided.

pointer

A pointer has two subfields. The most significant 16 bits is the access control field. described in §6.1. The remaining 48 bits make up the address field. described in §6.2.

instruction

Instructions, composed of operations, are described in §3.

stream status word

A stream status word (SSW), contains status and control information for the instruction stream in its upper halfword and a program counter in the lower halfword. It is described in §2.1.

resource counter

The processor counts interesting events for accounting and performance monitoring. They are described in §10.

Several data types are derived from type Boolean, a single-bit unsigned type, where 0 is *false* and 1 is *true*. The name of each derived type is a mnemonic to help interpret what the bit controls when active—namely when it is *set*, is *true*, or is assigned 1, all of which are equivalent terms. For example, a variable of type Flag notes that an exception has occurred if it is set; a variable of type SignBit indicates a negative number if it is set.

Several other types are implicitly derived from type Uns, an unsigned datum of length at most 64 bits, as shown below:

type	width(bits)	base type	description
Reg	5	Uns	a register number
ProgramAddrUns	32	Uns	a ProgramAddress structure treated as an Uns: see §2.1
PageNumber	20	Uns	a virtual program page address
ProgFrame	17	Uns	a physical memory offset
DataAddrUns	48	Uns	a DataAddress treated as an Uns; see §6.1
DataSegment	20	Uns	a virtual data segment number
SegmentOffset	15	Uns	an offset into a virtual data segment
DataFrame	19	Uns	a physical memory offset specified as a frame number or a physical memory frame number

1.3 Storage Classes

Each stream has available a number of different kinds of storage.

- There is a large amount of memory, all of it potentially available to any stream on any processor in the system. Data memory units adjacent to the referencing stream's processor have relatively low latency. This adjacent data memory is referred to as "local" and is currently used only to store instructions, data maps, and program maps for the local processor. Most data memory accesses are distributed across the entire system. The part of data memory that stores instructions for its processor, is sometimes called "program memory". Every word in data memory has a four-bit access state, which modifies the behavior of memory references to any part of the word: see §6.1.

- The 31 general-purpose registers are used as the sources and destination for almost all operations. Register 0 always reads as 64 bits of 0, and values written into it are discarded.
- The stream status word (SSW) contains condition codes, the trap mask, the mode, and the program counter. The SSW is described in §2.1.
- The eight target registers contain program addresses and are used as arguments for branch operations; Target 0 points to the trap handler. See §2.2.
- The exception register flags the exception(s) that have been detected and raised. A raised exception will cause a trap if the trap is not disabled by the appropriate bit in the trap mask of the stream status word. The exception register also contains the register poison flags. See §9.1.
- The result code register describes exceptional result values from the function units: see §9.1.
- The trap registers are used by the trap handler to save the state of the trapping stream. The trap registers are described in §9.2.

1.3 Storage Classes

Chapter 2: Streams

Each physical processor supports a variable number of instruction streams. or streams for short. Each stream appears to be (and is programmed like) a wide-instruction RISC processor. The processor hardware selects streams for execution and executes a single instruction from each in turn. Streams are allocated, created, and destroyed dynamically; the active streams are multiplexed by the processor hardware onto a single set of pipelined functional units.

Streams may be active or idle. An active stream competes with other streams to issue instructions, while idle streams do not. A stream is activated and initialized with a skeleton execution environment by the unprivileged STREAM_CREATE operation. Unprivileged STREAM_RESERVE operations are used to reserve a number of idle streams for subsequent activation by STREAM_CREATE. The STREAM_QUIT operation returns a stream that executes it to the idle state.

A stream executes at one of four privilege levels: user, supervisor, kernel, or IPL. The privilege level of a stream determines the operations it may execute and the kinds of memory access it is permitted. Levels are described further in §8.1.

Each active stream in a processor belongs to one of sixteen protection domains. A protection domain has registers that limit the number of streams it can contain and define the memory accesses available to its streams. Protection domains are described in more detail in §8.2.

2.1 Stream Status Word

The stream status word (ssw) is shown below. The ssw contains the condition codes from the most recent four "_TEST" operations; a trap mask which selectively disables traps from raised exceptions; a mode field describing how arithmetic, memory references, and lookahead are to be done; and a program counter containing the address of the instruction being executed.

Bits	Wd	Field Name	Type	Description
StreamStatus Word: Condition Vector				
63–61	3	cc_3	CondCode	condition code cv_3: result from fourth most recent _TEST operation; see §4
60–58	3	cc_2	CondCode	condition code cv_2: result from third most recent _TEST operation; see §4
57–55	3	cc_1	CondCode	condition code cv_1: result from second most recent _TEST operation; see §4
54–52	3	cc_0	CondCode	condition code cv_0: result from most recent _TEST operation; see §4

Stream Status Word

StreamStatusWord: Trap Mask

51	1	hardware_trap_-disable	Boolean	disable hardware traps
50	1	system_trap_-disable	Boolean	disable system traps
49	1	domain_signal_-trap_disable	Boolean	disable domain signal traps
48	1	user_trap_disable	Boolean	disable user traps
47–45	3	0		*reserved*
44	1	float_invalid_-trap_disable	Boolean	disable float invalid trap
43	1	float_zero_div_-trap_disable	Boolean	disable float zero divide trap
42	1	float_overflow_-trap_disable	Boolean	disable float overflow trap
41	1	float_underflow_-trap_disable	Boolean	disable float underflow trap
40	1	float_inexact_-trap_disable	Boolean	disable float inexact trap

StreamStatusWord: Mode

39	1	0		*reserved*
38	1	ssw_override	Boolean	disables all traps, lookahead, and the instruction counter; allows some memory operations to retry forever: see §9.2
37	1	spec_load_enable	Boolean	allows loads to be speculative; see §6.4
36	1	unaligned_data_-enable	Boolean	prevents unaligned data from raising the data_alignment exception: see §6.1
35	1	lookahead_disable	Boolean	disables lookahead, so that each memory operation finishes before the next instruction is issued; see §3.1.
34	1	count_disable	Boolean	disables the instruction counter; see §10
33–32	2	round_mode	RoundMode	floating-point rounding mode; see §5.2

StreamStatusWord: Program Counter

31–0	32	pc	ProgramAddrUns	the program counter

The field "pc", shown here as a type ProgramAddrUns, is actually a structure of type ProgramAddress, used in program address translation; see §7.1.

2.2 Branches and Targets

There are two major families of branch operations. The JUMP family is intended for general long-distance transfers including subroutine calls. The SKIP family adds a small positive offset to the program counter and is intended for the short forward transfers needed in if-then-else situations. The JUMP and SKIP families have variants for terminating lookahead if the branch is or is not taken: see §3.1.

Jumps are performed in two distinct operations. First, a TARGET operation loads a target register with a program address. Second, the JUMP operation is executed. conditionally setting the ssw.pc to the contents of the specified target register. Separating these two concerns lets the processor prefetch instructions down an execution path that may be taken in the future. Loading target registers with invalid addresses will not raise an exception unless and until the target register is used in a successful JUMP operation.

There are eight target registers. Each target register contains a program counter; see §2.1. Target register T0 is reserved for the address of the trap handler. It is automatically exchanged with the ssw.pc on a trap, and can be written by unprivileged streams (unless the "priv_t0" bit in the program state of the protection domain prohibits it). When a target register is loaded, the program cache attempts to prefetch the line containing the new address; see §7.2.

Chapter 3: Instructions

Every instruction is 64 bits long, and generally contains four fields describing lookahead, an M-operation, an A-operation, and a C-operation. These fields are shown here.

Bits	Wd	Field Name	Type	Description
Operation				
63–61	3	la	Uns	lookahead
60–47	14	Mop	Uns	M-operation
46–21	26	Aop	Uns	A-operation
20–0	21	Cop	Uns	C-operation

The lookahead field is used to control M-unit operation overlap and is described in §3.1. In general, an M-unit operation (M-operation) accesses memory in some way, an A-unit operation (A-operation) performs arithmetic, and a C-unit operation (C-operation) is primarily responsible for control flow. The C-operation can also do some arithmetic operations, exclusive of multiplication. Nearly every arithmetic operation that can be done in a C-operation can also be done by an A-operation.

Some operations are encoded by combining multiple operation fields. For example, an MC-operation such as INT_LOAD_DISP uses both the M- and C-operation fields. STREAM_CREATE and STREAM_QUIT are MAC-operations.

The operations in an instruction are decoded in parallel. If any of them is invalid, either because it is a privileged operation at the current protection level or it is an illegal operation encoding, a privileged operation exception is raised, and no part of the instruction is issued.

The decoded operations are executed in parallel. All operands for all operations in the instruction are read before any result is written. Results are written in an implementation-dependent order, so if more than two operations in an instruction write to the same destination register, the resulting value is undefined. Thus, such an instruction is illegal. Once instruction execution is begun the destination registers are always written, regardless of whether or not the operation later raises an exception or traps.

The program counter (PC) follows the same rule for reading and writing as the operands. The PC is read when the instruction is issued and is written when the instruction completes. The written value is either an incremented value for normal sequential flow or a new value from a branch.

The individual operations are described in §11.

3.1 Lookahead

The lookahead field is a three-bit unsigned integer that the code generator must guarantee to be less than or equal to the minimum number of instructions that the stream might execute before encountering one that depends on the current M-, MC-, or MAC-operation. The maximum possible lookahead value is seven. If there is no such operation, the code generator should set the lookahead to the maximum of seven. If the code generator is ignorant of the relevant dependences, the lookahead may be set to zero. The lookahead must take into account all branch paths that are lookahead-enabled, as described below.

An instruction *J* depends on an M-unit operation (M-, MC-, or MAC-operation) at a prior instruction *I* if any operation in *J* uses or defines a register or a part of a register implicitly or explicitly defined by the M-unit operation in *I*. In addition, an instruction *J* depends on an M-unit operation at a prior instruction *I* if the M-unit operation in *J* references some of the same memory referenced in *I* and the memory is modified by either or both of *I* and *J*. These definitions are manifestations of standard data dependence rules.

The lookahead field supplies the hardware with an upper bound on the number of additional instructions that may begin execution before the current M-unit operation is finished. For example, if lookahead is zero throughout a program, then the processor will finish each M-, MC-, or MAC-operation before starting the next instruction. Lookahead can be disabled to get the same effect by setting the mode bit field "lookahead_disable" in the SSW.

Branch paths are determined by branching operations and their corresponding skip amounts or target registers. All conditional branch operations have variants that disable lookahead on one of the two paths. The "_SELDOM" branch operations (JUMP_SELDOM, SKIP_SELDOM) disable lookahead when the transfer is taken, and the "_OFTEN" branch operations (JUMP_OFTEN, SKIP_OFTEN) disable lookahead when the transfer is not taken. The effect of disabling lookahead is to require all outstanding memory references to complete before the next instruction is allowed to execute.

Chapter 4: Condition Codes

Many operations have alternate versions (with "_TEST" appended to the mnemonic) that generate a condition code in addition to a value in a register. The eight possible condition code values and their default meanings are shown below, where 0, p, and n stand for zero, a positive integer, and a negative integer, respectively.

Name	Value	Meaning	Examples
CondCode			
COND_ZERO_NC	0	Zero, no carry	$0 = 0 + 0$
COND_NEG_NC	1	Negative, no carry	$n = p + n, n = p - p$
COND_POS_NC	2	Positive, no carry	$p = p + p, p = p - n$
COND_OVFNAN_NC	3	Overflow/NaN, no carry	$n = p + p, n = p - n$
COND_ZERO_C	4	Zero, carry	$0 = n + p, 0 = n - n$
COND_NEG_C	5	Negative, carry	$n = n + n, n = n - p$
COND_POS_C	6	Positive, carry	$p = n + p, p = n - n$
COND_OVFNAN_C	7	Overflow/NaN, carry	$p = n + n, p = n - p$

Each newly generated condition code is inserted as cv_0 at the low end of the four-element condition vector cv associated with the stream; the existing codes shift over and the old value of cv_3 is lost. If multiple operations in the same instruction generate condition codes (because there are multiple "_TEST" suffixes), then the condition code from the C-operation is inserted first, followed by the condition code from the A-operation.

After integer arithmetic operations, the condition code describes the sign of the result in the obvious way unless overflow has occurred, in which case the result sign is negative if and only if there was no carry. Some integer and bit operations—such as INT_MAX and BIT_RIGHT_ONES— generate the carry bit in a nonstandard way; for these operations overflow/NaN is not generated, and the condition code still accurately reflects the sign of the result.

After floating-point operations, the condition code describes the result in a way compatible with IEEE Standard 754. See §5 describing floating-point arithmetic.

A condition mask, shown as *cond* in the operation descriptions, describes a set of condition code values by summing the powers of two corresponding to the codes in the set, typically to determine whether a branch should take place. A *cond* can describe any combination of condition codes. For example, the condition mask named IF_EQ (if equal) describes codes 0 and 4, so it has the value $2^0 + 2^4$, which is 0x11 or 11_{16}.

Most of the important condition masks have one or more names. The named condition masks are shown below.

Name	Value	After (SUB_TEST x y z)
CondMask: Manifest		
IF_ALWAYS	0 1 2 3 4 5 6 7	always
IF_NEVER		never
CondMask: Equality		
IF_EQ	0 4	$y = z$ (integer, unsigned, float)
IF_ZE	0 4	$x = 0$ (integer. unsigned, float)
IF_F	0 4	$x = 0$ (logical)
IF_NE	1 2 3 5 6 7	$y \neq z$ (integer, unsigned, float)
IF_NZ	1 2 3 5 6 7	$x \neq 0$ (integer, unsigned, float)
IF_T	1 2 3 5 6 7	$x \neq 0$ (logical)
CondMask: Integer Comparison		
IF_ILT	1 5 7	$y < z$ (integer)
IF_IGE	0 2 3 4 6	$y \geq z$ (integer)
IF_IGT	2 3 6	$y > z$ (integer)
IF_ILE	0 1 4 5 7	$y \leq z$ (integer)
IF_IMI	1 3 5	$x < 0$ (integer)
IF_IPZ	0 2 4 6 7	$x \geq 0$ (integer)
IF_IPL	2 6 7	$x > 0$ (integer)
IF_IMZ	0 1 3 4 5	$x \leq 0$ (integer)
CondMask: Unsigned Comparison		
IF_ULT	1 2 3	$y < z$ (unsigned)
IF_UGE	0 4 5 6 7	$y \geq z$ (unsigned)
IF_UGT	5 6 7	$y > z$ (unsigned)
IF_ULE	0 1 2 3 4	$y \leq z$ (unsigned)
CondMask: Float Comparison		
IF_FLT	1 5	$y < z$ (float)
IF_FGE	0 2 4 6	$y \geq z$ (float)
IF_FGT	2 6	$y > z$ (float)
IF_FLE	0 1 4 5	$y \leq z$ (float)
CondMask: Other Tests		
IF_IOV	3 7	x overflowed (integer)
IF_FUN	3 7	y and z are unordered (float)
IF_CY	4 5 6 7	carry
IF_NC	0 1 2 3	no carry

CondMask: Specific Conditions		
IF_0	0	Zero. no carry
IF_1	1	Negative, no carry
IF_2	2	Positive. no carry
IF_3	3	Overflow/NaN, no carry
IF_4	4	Zero, carry
IF_5	5	Negative, carry
IF_6	6	Positive, carry
IF_7	7	Overflow/NaN, carry

4.1 Select Operations

The SELECT_ operations use three-bit encodings to specify one of eight of the most common condition masks. SELECT_INT uses a mask *IntSelect* that encodes integer and unsigned comparisons as shown below. Additional selects can be realized by reversing the arguments u and v of the SELECT_INT operation itself.

Name	Value	After (SUB_TEST x y z)
IntSelect		
SEL_CY	0	carry
SEL_EQ	1	$y = z$ (integer, unsigned. float)
SEL_IGT	2	$y > z$ (integer)
SEL_IGE	3	$y \geq z$ (integer)
SEL_UGT	4	$y > z$ (unsigned)
SEL_UGE	5	$y \geq z$ (unsigned)
SEL_IPL	6	$x > 0$ (integer)
SEL_IPZ	7	$x \geq 0$ (integer)

The SELECT_FLOAT operation uses the encoding *FloatSelect* as shown below.

Name	Value	After (FLOAT_MIN_TEST x y z)
FloatSelect		
SEL_FLT	2	$y < z$ (float)
SEL_FLE	3	$y \leq z$ (float)
SEL_FGT	4	$y > z$ (float)
SEL_FGE	5	$y \geq z$ (float)
SEL_FUN	6	y and z are unordered (float)

An IntSelect or FloatSelect enumeration describes the same condition code set as the identically suffixed CondMask.

Chapter 5: Floating-point Arithmetic

5.1 Floating-point Formats

The IEEE Standard 754 floating-point double basic format (64 bit) and single basic format are supported. This is the structure of a normal 64-bit floating-point number:

Bits	Wd	Field Name	Type	Description
Float64				
63	1	sign	SignBit	sign bit
62–52	11	exponent	Uns	biased exponent
51–0	52	fraction	Uns	fraction

This is the structure of a normal 32-bit floating-point number:

Bits	Wd	Field Name	Type	Description
Float32				
31	1	sign	SignBit	sign bit
30–23	8	exponent	Uns	exponent
22–0	23	fraction	Uns	fraction

Doubled precision addition. subtraction, multiplication, and conversion operations to and from the single precision IEEE 754 format and both signed and unsigned integer formats are supported directly. Division and square root are accomplished with the help of iterative computation primitives that use a special floating-point format providing extra significand precision:

Bits	Wd	Field Name	Type	Description
SpecialFloat64				
63	1	sign	SignBit	sign bit
62–53	10	exponent	Uns	biased exponent
52–0	53	fraction	Uns	fraction

The SpecialFloat64 exponent is biased by 510, so that the true exponent is the biased exponent minus 510.

All operations conform to the applicable IEEE standard.

Floating-point comparison operations set the condition code to indicate whether the operands are equal, greater, less, or NaN. The carry bit indicates the second operand ($v \vee z$) is a NaN. The float_invalid exception is never raised by FLOAT_MIN or FLOAT_MAX. When the compare is performed by a FLOAT_CMP_TEST, float_invalid is raised when the operands are unordered. Thus, IEEE 754 tests which do not raise an exception on unordered operands, such as a test for equality, should

Special 64-bit Floating-point Format

be implemented using FLOAT_MIN_TEST. Tests for inequalities such as greater than should use FLOAT_CMP_TEST to properly handle unordered operands.

Support for fast doubled precision arithmetic is provided. In doubled precision, a pair of 64-bit floating-point numbers is used to hold twice the significant digits and provide at least twice the precision of ordinary 64-bit floating point. There are provisions to compute the doubled precision sum, difference, and product efficiently. See the doubled precision programming examples in §12.4.

5.2 Rounding

Unless explicitly specified otherwise in an operation description, rounding is performed according to the rounding mode stored in field "round_mode" in the SSW. The rounding modes are shown here.

Name	Value	Meaning
RoundMode		
RND_NEAR	0	round to nearest
RND_CHOP	1	round toward zero
RND_FLOOR	2	round toward $-\infty$
RND_CEIL	3	round toward ∞

Rounding is explicitly specified in some convert operations, such as FLOAT_CEIL, INT_CHOP, and UNS_FLOOR.

5.3 Floating-point Exceptions

Floating-point exceptions are raised as a side effect of operation completion. The destination register of the operation is set in accordance with the IEEE Standard.

Besides the 64-bit result in the destination register, a floating-point exception records the destination register number and a four-bit floating-point result code in the result code register. A nonzero result code indicates that the destination register contains an exceptional value and summarizes that value. Floating-point result codes are described in §9.1. Note that a zero destination register will not allow the exceptional value to be saved.

In conformance with IEEE Standard 754, an invalid operation exception is raised (and a trap potentially taken) when a conditional test operation encounters a NaN when performing an inequality test as described in §5.2.

If overflow or underflow traps are disabled, then overflow delivers infinity or a maximum magnitude floating-point value, depending on the rounding mode, and underflow delivers a denormalized result. When floating-point overflow or underflow traps are enabled, the result in the destination register is the same as the masked response, so that the trap handler may report the program state and resume execution. Note that underflow is only raised when the result is inexact and subnormal, whether the underflow trap is enabled or not. The check for subnormal is before rounding, so the final result may actually be normalized (due to rounding). A float_zero_divide exception always returns a properly signed infinity. A float_extension exception returns the argument to the operation (they are all unary) so that the trap handler may easily locate the value and complete the operation.

A float_invalid exception generates a NaN value in accordance with the IEEE standard. A NaN generated by an operation describes the reason for the exception using the enumeration below. The appropriate code is stored in the low three bits of the fraction.

Name	Value	Meaning
NaNResultCode		
NAN_ZERO_MUL_INF	1	Zero times infinity
NAN_INF_SUB_INF	2	Magnitude subtraction of infinities
NAN_ZERO_DIV_ZERO	4	Zero divided by zero
NAN_INF_DIV_INF	5	Infinity divided by infinity
NAN_SQRT_NEG	6	Square root of negative number

The high-order fraction bits of a NaN are zero; these bits are Bits 51–3 for a normal 64-bit floating-point number, and Bits 22–3 for a 32-bit floating-point number. The destination register (t or x) is stored in the result code register, so that the NaN may be examined by the trap handler for diagnosis or continuation.

There are no signaling NaNs, but data trap bits provide a more comprehensive mechanism; see §6.1.

Chapter 6: Data Memory

A Tera system has either two or four data memory units per processor. When four units per processor are configured, the additional two units are referred to as "expanded data memory". Data is accessed by LOAD, STORE, FETCH_ADD, and STATE operations. This chapter describes what is stored in data memory, the semantics of accessing it, the address translation mechanism, and finally the internal state of the M functional unit. The M-unit can simultaneously process up to eight pending requests for data memory access by each stream.

6.1 Data Memory Access

Every data memory cell contains a 64-bit value and a four-bit access state. The value in a memory cell can be addressed as a word, 2 halfwords, 4 quarterwords, or 8 bytes. The order of bytes in quarterwords, quarterwords in halfwords, and halfwords in words is "big-endian", i.e. packed so that addresses increase as significance decreases. Thus the word at address A, read from left to right, most significant bit to least, contains bytes with addresses A, $A+1, \ldots A+7$; quarterwords with addresses A, $A+2$, $A+4$, and $A+6$; and halfwords with addresses A and $A+4$.

The access state modifies the behavior of memory references to the word or partial word contained in the cell. It has this structure:

Bits	Wd	Field Name	Type	Description
AccessState				
3	1	full	Boolean	full/empty bit
2	1	forward_enable	Boolean	forward enable
1	1	trap1_enable	Boolean	data trap 1 enable
0	1	trap0_enable	Boolean	data trap 0 enable

The operations STATE_LOAD, STATE_STORE, and STATE_LOCK are respectively used to load, store, and lock the access state.

Operations always access data memory relative to a pointer. The semantics of memory access are determined by an access control field in this pointer, possibly overridden by an access control field in the operation, and by the access state of the addressed memory cell(s). Briefly, the access can be forced to wait until the cell is either empty or full, a data blocked exception can be raised in response to load or store accesses to the cell, and a memory cell can forward accesses to another memory cell.

A pointer has two parts: an access control part, which modifies access through the pointer, and an address part. The fields in a pointer are as follows:

6.1 Data Memory Access Pointer with Access Control Field

Bits	Wd	Field Name	Type	Description
Pointer: access control				
63	1	0		*reserved*
62	1	fwd_disable	Boolean	forwarding disable
61–60	2	fe_control	FullEmptyControl	full/empty control
59	1	trap1_store_-disable	Boolean	data trap 1 disable on store
58	1	trap1_load_disable	Boolean	data trap 1 disable on load
57	1	trap0_store_-disable	Boolean	data trap 0 disable on store
56	1	trap0_load_disable	Boolean	data trap 0 disable on load
55–48	8	0		*reserved*
Pointer: address				
47–0	48	address	DataAddrUns	data memory address

The field "address", shown here as a type DataAddrUns, is actually a structure of type DataAddress. used in data memory address translation; see §6.2.

The value in the field "fe_control" is of type FullEmptyControl, described here. In the description of the load and store behaviors, the term "waits for empty (full)" means that the operation waits until the field "full" in the memory cell's full bit becomes false (true); the term "sets empty (full)" means that the field "full" is set to false (true).

Name	Value	Behavior
FullEmptyControl		
FE_NORMAL	0	LOAD loads; STORE stores and sets full
FE_FUTURE	2	LOAD waits for full, then loads; STORE waits for full, then stores
FE_SYNC	3	LOAD waits for full, then loads and sets empty; STORE waits for empty, then stores and sets full

Some memory operations encode an access control operand, abbreviated *ac*, that supersedes the pointer's access control specification. The operation access control structure is shown here:

Bits	Wd	Field Name	Type	Description
OperationAccessControl				
4	1	fwd_disable	Boolean	forwarding disable
3–2	2	fe_control	FullEmptyControl	full/empty control
1	1	trap1_disable	Boolean	data trap 1 disable
0	1	trap0_disable	Boolean	data trap 0 disable

Memory reference operations first add the address field from a pointer held in register *s* to an optional scaled offset. Addition is done modulo 2^{48}. The offset is derived from either another register *y* or from an unsigned literal displacement *disp* in the instruction and is then scaled (multiplied)

Operation Access Control Field

by the size in bytes of the addressed object. The length of the *disp* field varies so that the scaled offset covers the same set of memory locations independent of object size. This sum is the effective address of a word, halfword, quarterword, or byte in memory.

Then, the effective address is checked against the map limit for this domain. If the limit is exceeded, a data map limit exception is raised.

Unless the field "unaligned_data_enable" is set in the ssw, a data alignment exception will be raised when an effective address presented to memory is not a multiple of the number of bytes in the addressed object.

At this point, the data map entry is consulted. If the current privilege level of this stream is insufficient for the map's protection level for this type of operation, a data protection level exception is raised. Otherwise, the segment offset in the effective address is checked against the segment limit in the map entry. If the limit is exceeded, a data segment limit exception is raised.

Next, a data blocked exception is raised if a data trap bit is enabled in the addressed word and the corresponding data trap disable bit is clear in *ac* or in *s* when *ac* is not present.

Forwarding is examined and handled next. If *ac* is present in this operation, its forward disable bit is used; otherwise that of *s* is used. If the selected forwarding disable bit is clear and the forward bit is enabled in the addressed word, then the cell may contain a forwarding pointer rather than the data itself. If the cell is forwarded and empty then the operation is retried later; the interpretation is that the forwarding pointer is locked. If the cell is full then the value in the cell is used as an effective word address for another memory access. No registers are modified in this process. The relative word position of a partial word access is unchanged; the three least significant bits of the forwarded effective address are copied from the original address. Data traps at forwarded locations are processed as usual; the data trap disable bits in effect are the original data trap disable bits. The forwarding disable bit in effect is the original (clear) forwarding disable bit. The full/empty control bits are taken without modification from *ac* or from *s* when *ac* is not present.

Finally, synchronization is handled. The full bit in the addressed word is processed in conjunction with the full/empty control bits from *ac*, or *s* when *ac* is not present.

No memory full bit testing occurs if full/empty control is FE_NORMAL; in this case load operations fetch the value of the addressed word or partial word into register *r*, and store operations store the contents of *r* into the addressed word or partial word and set it full.

If full/empty control is FE_FUTURE or FE_SYNC, then the memory full bit is tested. If its state is the one waited for, then the load or store of the value occurs, and the memory full bit is changed if full/empty control was FE_SYNC. Otherwise, the operation is retried later.

When the operation is retried it starts over with the original address (before any forwarding). If the total number of memory cell accesses due to forwarding and retrying exceeds the retry limit in the data state descriptor, a data blocked exception is raised and the operation is aborted. Retries may also be caused by network contention, translation stalls, and other miscellaneous hardware events. When ssw_override mode is set, all memory operations except synchronizing loads, stores, and int_fetch_adds will retry forever and will not raise the data blocked exception.

6.2 Data Memory Address Translation

Data Memory Address

6.2 Data Memory Address Translation

Data memory addresses are found in Bits 47–0 of pointers. A data memory address has this structure:

Bits	Wd	Field Name	Type	Description
DataAddress				
47–28	20	data_segment_-number	DataSegment	data segment number
27–13	15	data_segment_-offset	SegmentOffset	data segment offset
12–3	10	data_frame_offset	Uns	data frame offset
2–0	3	byte_offset	Uns	byte offset

A complete data memory address is 48 bits long, potentially addressing 256 Terabytes of memory. However, only 42 bits of the address are currently implemented, and the high-order six bits of the data segment number must be zero. The implemented data address space is consequently 4 Terabytes. This space is partitioned into 16K segments, each of which can vary in size from 8 Kbyte to 256 Mbyte in 8 Kbyte increments.

Data memory address translation proceeds in five logical steps. The translation logic block diagram is shown in Figure 6.1. First, the data segment number is validated. Second, the data segment map is accessed, yielding a data segment map entry. Third, the protection level is checked and the segment address is limited and relocated using values in the map entry. This yields a logical address in two parts, a logical unit number and logical unit offset. Fourth, the data frame offset is transformed so that memory references are scrambled, yielding the logical frame offset. Fifth, the logical address is distributed to spread references among the logical units (the memory resources). These steps are now described in more detail.

The first logical step is to validate the data segment number and select the data map to use for translation. If the protection domain's data map limit from the protection domain's data state descriptor is less than the data segment number, a data map limit exception is raised. Otherwise, the resulting segment number and domain number are sent to the data segment map. At this point, the alignment requirements for the selected operation are checked against the effective address. If the reference is unaligned and field "unaligned_data_enable" of the SSW is clear, the data alignment exception is raised.

The second step in the translation reads a data map entry from the data segment map. Each entry has the structure shown here:

Data Map Entry

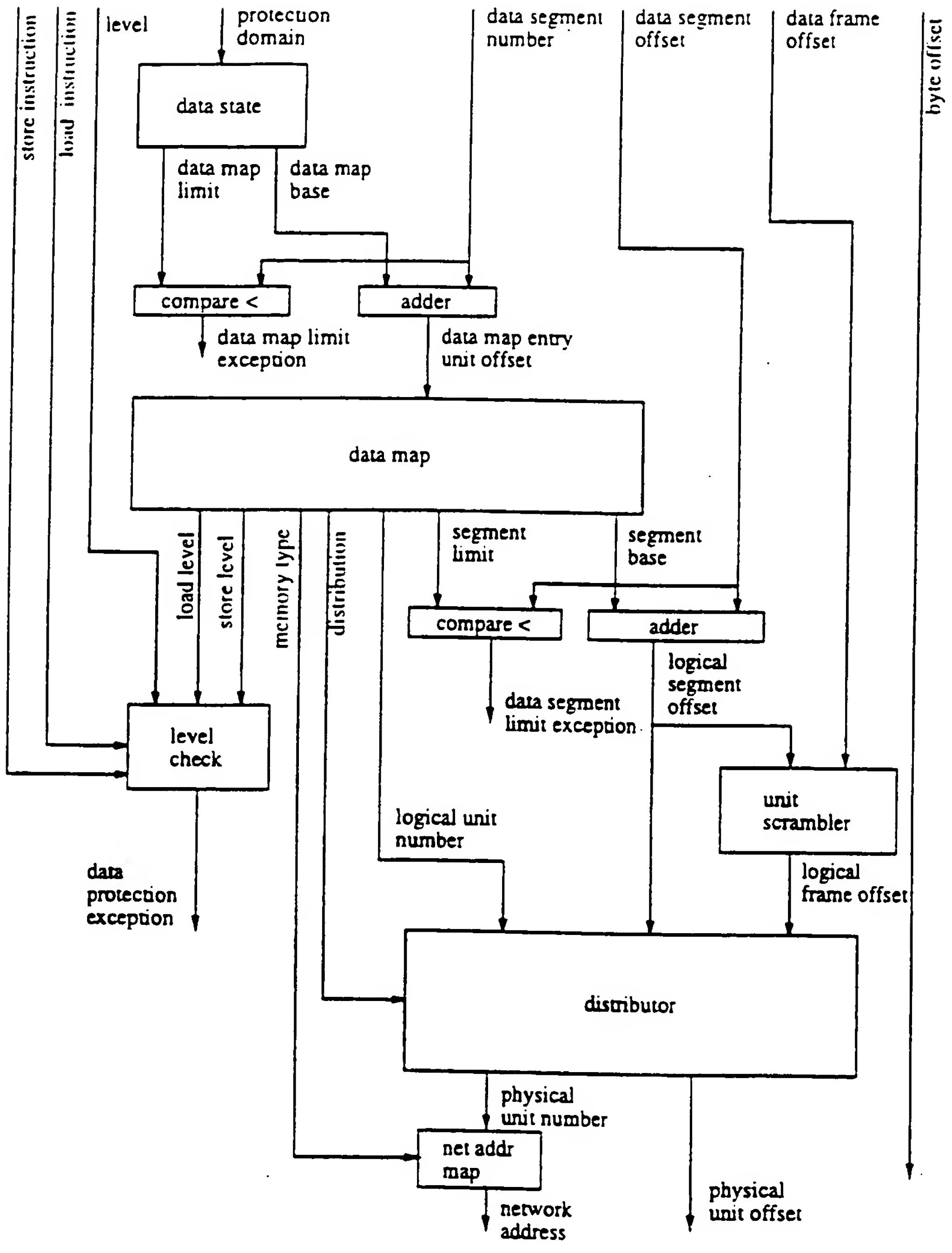

FIGURE 6.1: Data Mapping Logic Block Diagram

6.2 Data Memory Address Translation

Bits	Wd	Field Name	Type	Description
DataMapEntry				
63–62	2	store_level	Level	minimum store protection level
61–60	2	load_level	Level	minimum load protection level
59–57	3	0		*reserved*
56	1	stall	Boolean	stall references to this entry
55	1	locked	Boolean	lock this map entry into TLB
54	1	distribution_-enable	Boolean	distribution
53–52	2	memory_type	Resource	select data memory, expanded data memory, or IOP units
51–48	4	0		*reserved*
47–40	8	logical_unit	Uns	logical unit number
39	1	0		*reserved*
38–24	15	segment_limit	SegmentOffset	segment limit
23–19	5	0		*reserved*
18–0	19	segment_base	DataFrame	segment base

This map is stored in local data memory, starting at the data map base for the given domain (see §8.4). To speed translation. a translation lookaside buffer (TLB) caches the map entries. Coherency is maintained by flushing modified entries from the cache using the DATA_MAP_FLUSH operation. The desired entries to flush are specified with a domain data address, which combines the domain and data address to index the data map. Entries with field "locked" set will only be evicted from the cache by a DATA_MAP_FLUSH operation which matches the entry. The implemented data map cache contains 512 entries. with four-way associativity. Within each set, entries are replaced using a least-recently-used policy. To reduce contention between domains, the domain number times eight is exclusive-or'ed with the set index before addressing the TLB. The DATA_MAP_FLUSH_ANY operation treats the TLB as direct mapped, using the low two bits of the tag as the set index, using a domain data TLB address. To flush all entries for a domain from the cache, each entry must be accessed. The flush addresses must sequence through all possible values of the set_index and set_number. The domain data address and domain data TLB address structures are shown below:

Bits	Wd	Field Name	Type	Description
DomainDataAddress				
63–60	4	domain	Uns	the domain to which this data address pertains
59–42	18	00000		*reserved*
41–35	7	tag	Uns	data TLB tag
34–28	7	set_number	Uns	data TLB set number
27–0	28	segment_offset	Uns	untranslated bits

Domain Data Address

Bits	Wd	Field Name	Type	Description
DomainDataTLBAddress				
63–60	4	domain	Uns	the domain to which this data address pertains
59–42	18	00000		*reserved*
41–37	5	tag	Uns	data TLB tag
36–35	2	set_index	Uns	data TLB set index
34–28	7	set_number	Uns	data TLB set number
27–0	28	segment_offset	Uns	untranslated bits

The third translation step limits and relocates the segmented address. If the current privilege level of the stream storing (loading) data in this segment is less than the minimum store (load) protection level in field "store_level" (field "load_level") of the data map entry, then a data protection level exception is raised. Note that many load operations will need store privilege to properly update the access state. However, store operations are implicitly given load privilege to properly follow access control waiting or trapping. That is, the store protection level is assumed to be no higher than the load protection level.

The segment limit, field "segment_limit", is compared with Bits 27–13 of the data address. If the segment limit is smaller, then a data segment limit exception is raised.

If the field "stall" is set, then the operation is returned to the retry pool and tried again later. This forced retry allows the supervisor to perform some memory management operations without stopping all activity in the domain. Note that the PROBE operation is not affected by field "stall", as its result is determined by the earlier segment limit check.

Otherwise, Bits 27–13 of the data address, extended with zeros on the left to 19 bits, are added to the segment base field "segment_base" in the data map entry. The sum is sent to the address scrambler as the logical segment offset.

The fourth step scrambles the Bits 21–3 of the data address, producing the logical frame offset. The concatenation of the logical segment offset and the data frame offset is treated as a 29-element vector in $GF(2)^{29}$. ($GF(2)$ is the field with elements 0 and 1, and as its multiply operation, and exclusive-or as its addition operation; $GF(2)^{29}$ is the vector space of dimension 29 over this field.) The vector is scrambled by multiplying it by the unit scrambling matrix, a fixed 29-by-19 bit matrix whose low-order 19-by-19 bit submatrix is invertible. This multiply yields the logical frame offset.

The scrambling matrix is chosen to make any sequence of constant-stride addresses spanning a length $s < 2^{29}$ generate a nearly uniform distribution in the logical frame offset, which in turn generates a nearly uniform distribution in the physical unit number field and the low-order bank bits of the unit offset. Appendix C.1 specifies the matrix and the inverse of the low-order 19-by-19 submatrix.

The distributor takes the concatenation of the logical unit number (field "logical_unit" from the data map entry), the logical segment offset, and the logical frame offset as its logical address. The next step distributes this logical address to control physical locality of reference. The distributor transforms the logical address into a physical address consisting of an eight-bit physical unit number and a 29-bit physical unit offset.

The distribution bit (field "distribution_enable") in the data map entry allows references to be spread over all p memories in the system, rather than staying within one memory unit. Here, p is set via the scan system and usually matches the number of processors, making p a power of two. However, in the presence of a faulty memory, p may also be a power of two less one. Distribution divides the logical unit address by p (or $p+1$ with a faulty memory) to effectively right shift the low-order bits of the logical unit number into the high-order bit positions of the logical unit offset and replace them with the low-order bits from the logical frame offset. Thus, the remainder becomes the new unit number and the quotient the unit offset. With a faulty memory, hardware mapping will allow distribution to bypass a faulty resource. This mapping may be set differently for each resource class, so that a system can run with any one faulty normal data memory resource and any one faulty expanded data memory resource. Even when distribution is disabled, this mapping will be in effect, so that the logical unit space appears contiguous.

This scheme allows distribution across physical memory units under control of the data map entry. A data map entry with distribution enabled will address all usable physical memory units in the system, implying that only $2^{29}/p$ words are addressed in each unit as (word) addresses increase from 0 to $2^{29}-1$. Moreover, the logical unit number u appearing in a data map entry is required to be less than p when distribution is enabled. If u is too large, a data address unimplemented exception is raised.

The low-order three bits of the original address are the byte address of the datum within the addressed word and are copied without modification into the low-order three bits of the final unit offset.

The Tera MTA computer supports machine subsetting. This feature allows any power of two subset of a machine to appear to software as an independent machine. For example, a 16-processor machine could be split into two eight-processor machines. In such a case, the interconnection network need not be split, but can be shared. To support subsetting, a physical unit base register is set up to convert unit numbers from the 0 to $p-1$ range to the appropriate range in the actual machine.

The resulting address is then sent to the network for routing to a memory unit. The physical unit number and the field "memory_type" are used to construct the network address, which controls network routing. The memory type should be selected from the values in the following enumeration:

Name	Value	Behavior
Resource		
RES_DMEM	0	Expanded data memory
RES_IOM	1	Normal data memory
RES_IOP	2	I/O processor

The hardware supports an option which combines the normal and expanded data memory for global distribution. When that option is enabled, RES_DMEM selects the bottom 1 gigabyte per processor of the global memory pool and RES_IOM selects the top 1 gigabyte.

If the physical address is unimplemented, then a data protection exception is raised when the issuing operation completes. If the memory system detects an uncorrectable error such as a double-bit ECC error on the data loaded from memory, then a data hardware error exception is raised when the issuing operation completes. To help detect hot spots, successful loads which take an excessive amount of time to travel from the processor to the memory will raise a latency limit exception while performing the load. Synchronizing loads which retry are not subject to the limit until they succeed. This limit is set during system initialization. This implementation checks the limit with a granularity of 16 cycles. The compares are performed modulo 4096 cycles, so that a latency of 4112 would appear as a latency of 16.

6.3 M-unit Internal State

The M-unit processes memory requests that are generated by M-operations. The M-unit may have up to eight requests simultaneously pending for each stream. The M-unit completes each request asynchronously.

For each stream, the state of any failed M-operations is held in eight pairs of registers called the data control registers and data value registers. A trap handler can save these register pairs using the DATA_OPA_SAVE and DATA_OPD_SAVE operations and can later use them to retry the operation with DATA_OP_REDO. The values in these registers are now described in more detail.

The eight data control registers contain address and control information for up to eight memory reference operations in progress in the M-unit, due to lookahead. The M-unit writes one of these registers when a memory operation is initiated, and reads it as it (re)tries the reference. When no operations are in progress, the program may read them directly using the DATA_OPA_SAVE operation. Each of the data control registers contains a data control descriptor with the structure shown here.

Bits	Wd	Field Name	Type	Description
DataControlDescriptor				
63	1	0		*reserved*
62	1	fwd_disable	Boolean	forwarding disable
61–60	2	fe_control	FullEmptyControl	full/empty control
59	1	trap1_disable	Boolean	data trap 1 disable
58	1	trap0_disable	Boolean	data trap 0 disable
57–53	5	dest_reg	Reg	destination or source register
52–48	5	restop	RetryOpCode	rest of the operation code
47–0	48	address	DataAddrUns	byte address

The value in the field "restop" encodes the operation that failed and raised an exception. The high-order bit of the field "restop" is set if the operation was a load (more precisely, an operation that writes a register upon completion) and is cleared if the operation was a store. The RetryOpCode enumeration is shown here.

Name	Value	Meaning
RetryOpCode: Stores		
OPA_STOREB	0	STOREB
OPA_STOREQ	1	STOREQ
OPA_STOREH	2	STOREH
OPA_STORE	3	STORE
OPA_STATE_STORE	7	STATE_STORE
OPA_STORE_EMPTY	11	REG_STORE

RetryOpCode: Loads		
OPA_INT_LOADB	16	INT_LOADB
OPA_INT_LOADQ	17	INT_LOADQ
OPA_INT_LOADH	18	INT_LOADH
OPA_INT_FETCH_ADD	19	INT_FETCH_ADD
OPA_UNS_LOADB	20	UNS_LOADB
OPA_UNS_LOADQ	21	UNS_LOADQ
OPA_UNS_LOADH	22	UNS_LOADH
OPA_LOAD	23	LOAD
OPA_STATE_LOAD	24	STATE_LOAD
OPA_STATE_LOCK	25	STATE_LOCK
OPA_PROBE	26	PROBE
OPA_REG_LOAD	27	REG_LOAD
OPA_SCRUB_LOAD	29	SCRUB_LOAD
RetryOpCode: Internal Codes		
OPA_STREAM_CREATE	4	STREAM_CREATE
OPA_MAP_FLUSH	5	DATA_MAP_FLUSH
OPA_MAP_FLUSH_ANY	6	DATA_MAP_FLUSH_ANY
OPA_DATA_STATE_-RESTORE	9	DATA_STATE_RESTORE
OPA_STREAM_CATCH	12	STREAM_CATCH
OPA_DATA_OPD_SAVE	14	DATA_OPD_SAVE
OPA_DATA_OPA_SAVE	15	DATA_OPA_SAVE

The eight data value registers contain the data (if any) that the M-unit was attempting to write using the memory operation in the corresponding data control register. These registers are explicitly read by the program via the DATA_OPD_SAVE operation.

6.4 Speculative Loads

In speculative load mode, some data memory exceptions are deferred until the loaded value is used. In the usual circumstance, exceptional values are never used because the program (correctly) fails to use the prefetched data. Speculative loads allow data prefetching in iterative or recursive computations with data-dependent exit conditions.

When speculative loads are enabled (field "spec_load_enable" in SSW), a load with access control FE_NORMAL into register r that would otherwise raise a data alignment exception, a data segment limit exception, a data map limit exception, or a data protection level exception will instead place a data control descriptor (§6.3) in r and set the corresponding poison flag in the exception register(§9.1). Note that the field "dest_reg" of the data control descriptor is redundant. All other exceptions, including a data memory retry exception, are raised whether speculative loads are enabled or not. Whenever r is used as a destination register (even by a successful load, speculative or not) its poison flag is cleared. Use of a poisoned register r as a source operand raises a poison exception, except in REG_STORE, REG_MOVE, SELECT, and TRAP_RESTORE operations.

Chapter 7: Program Memory

A processor accesses instructions held in a program memory region of the data memory local to the processor. The term "program memory" is used to refer collectively to this region in data memory. This chapter describes what is stored in program memory, the semantics of accessing it, the address translation mechanism, and the instruction cache. Since it is part of data memory, each cell of program memory contains a four-bit access state and a word value, but this access state is ignored by the instruction fetching process. The value is a 64-bit instruction specifying up to three operations, one for each of the M-, A-, and C-units.

7.1 Program Memory Address Translation

Program addresses are 32 bits wide and are found in field "pc" occupying the low-order 32 bits of the stream status word and in the target registers. Only 25 bits of the program address space are implemented; the most significant seven bits must be zero. Thus the physical address space of the implementation allows for up to four gigawords of physical program memory but only 32 megawords are currently implemented. Program addresses are word rather than byte addresses and always address the data memory unit attached to the processor. A program address has the structure shown here:

Bits	Wd	Field Name	Type	Description
ProgramAddress				
31–12	20	prog_page_number	PageNumber	program page number
11–0	12	prog_page_offset	Uns	program page offset

Program memory address translation for all streams (regardless of level) uses the translation logic shown in Figure 7.1. Address translation proceeds in four steps. The first logical step is to limit the program page number and select the program map to use for translation. Second, the appropriate program page map for this page is accessed, yielding a program page map entry which is concatenated with the program page offset to form a logical unit offset. Finally, the logical unit offset is scrambled, resulting in a physical unit offset.

If the protection domain's program map limit (found in the protection domain's program state descriptor) is smaller than the program page number, then a program protection exception is raised. The selected map base is added to the page number to yield a unit offset into local program memory, where the page map entry is found.

The program map entry from the program page map has the structure shown below.

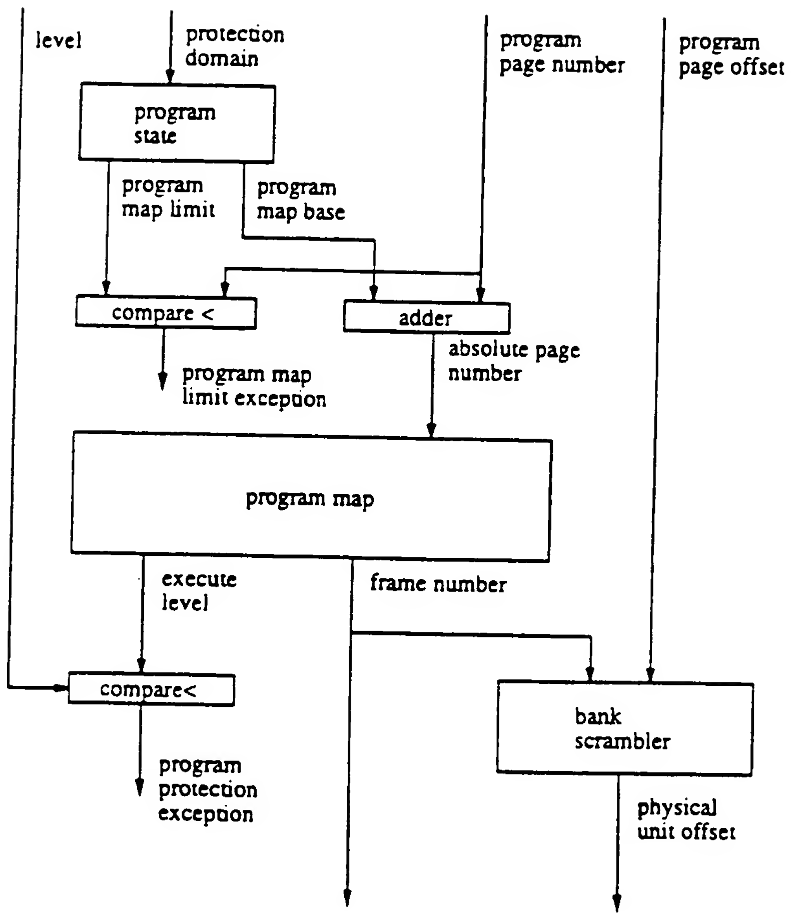

FIGURE 7.1: Program Mapping Logic Block Diagram

Bits	Wd	Field Name	Type	Description
ProgramMapEntry				
31–29	3	0		*reserved*
28–12	17	prog_frame	ProgFrame	frame number
11–2	10	0		*reserved*
1–0	2	exec_level	Level	execute level

The implementation restricts the field "prog_frame" to values from 0 to $2^{14}-2$. To reduce instruction fetch latency, a program map cache saves recently used map entries. This cache is kept coherent with program memory using explicit PROGRAM_MAP_FLUSH instructions. The desired entries to flush are specified with a domain program address, which combines the domain and program address to index the program map. This implementation provides a 128-entry program map cache backed by the L2 instruction cache (i.e., the L2 instruction cache holds program map entries as well as instructions). The program map cache is not associative; however, a small fully associative victim cache provides some tolerance for contention. To reduce contention between domains, the domain number times eight is exclusive-or'ed with the set index before addressing the TLB.

Bits	Wd	Field Name	Type	Description
DomainProgramAddress				
63–60	4	domain	Uns	the domain to which this data address pertains
60–33	28	0000000		*reserved*
31–0	32	address	ProgramAddrUns	program memory address

When accessing the program map cache, the program address is equated to the ProgTlbAddr structure below. To flush the mappings for a domain from the TLB, the flush addresses must sequence through all possible values of the field "set_number".

Bits	Wd	Field Name	Type	Description
ProgTlbAddr				
31–25	7	00		*reserved*
24–19	6	tag	Uns	program TLB tag
18–12	7	set_number	Uns	program TLB set number
11–0	12	frame_offset	Uns	untranslated bits

Since the TLB is backed by the L2 instruction cache, map entries must be flushed from L2 as well. The structure below is used for addressing map entries in the L2 cache. The PROGRAM_MAP_-FLUSH instruction automatically forwards a flush to the L2 cache, so that the sub-block of 64 map entries containing the referenced entry is flushed. The PROGRAM_MAP_FLUSH_ANY operation flushes the whole line containing the referenced entry. As before, flush addresses must sequence through all possible values of the field "set_number" in L2 as well.

Bits	Wd	Field Name	Type	Description
ProgL2Address				
31–25	7	00		*reserved*
24–21	4	set_number	Uns	set number for L2 cache
20–18	3	line_index	Uns	line index
17–12	6	subblock_index	Uns	subblock index
11–0	12	frame_offset	Uns	untranslated bits

When a stream attempts to issue an instruction, if the privilege level of the stream is not equal to the execution protection level field "exec_level" in the corresponding program page map entry, then a program protection exception is raised.

The program map yields a 17-bit frame number from field "prog_frame" which is concatenated to the low-order 12 bits of the program counter, forming a 29-bit logical unit offset.

The bits of this offset are scrambled exactly as for the data logical unit offset.

The resulting 29-bit logical unit offset is also the physical offset; there is no distributor in the program address translator. The logical unit offset is then sent to the attached data memory unit. If the address is unimplemented, then a program protection exception is raised when an attempt is made to issue an instruction which could not be fetched due to this exception. If the memory system detects an uncorrectable error (such as a double-bit ECC error on the data being retrieved from memory), then an uncorrectable program memory exception is raised.

Note that identical data and program logical offsets in the same data memory will address identical locations as long as the segment's data map entry field "distribution_enable" is clear.

7.2 The Instruction Cache

There is a primary and secondary instruction cache for each processor to reduce the required program memory bandwidth and improve latency. The caches are non-blocking, so that other streams may access the caches while a miss is being handled.

The primary cache (L1) holds 1024 instructions, organized in lines of four words. The 256 lines are two-way associative, so there are 128 sets. Lines are replaced using a least-recently-used policy. Both the primary and secondary caches are tagged with physical addresses. Physical addresses are mapped to the L1 cache as shown below.

Bits	Wd	Field Name	Type	Description
L1Address				
31–26	6	00		*reserved*
25–9	17	tag	Uns	physical L1 tag
8–2	7	set_number	Uns	L1 set number
1–0	2	line_index	Uns	L1 line index

The secondary cache (L2) holds one quarter million words of instruction and program map data. To reduce the number of tags, the data is organized into lines of 256 words, with eight sub-lines of 32 words each. The 1024 lines are four-way associative, so there are 256 sets. Here, lines are replaced

L2 Address

using a random policy. Note that 16 lines can contain a program frame. Physical addresses are mapped to the L2 cache as shown below.

Bits	Wd	Field Name	Type	Description
L2Address				
31–26	6	00		*reserved*
25–16	10	tag	Uns	physical L2 tag
15–8	8	set_number	Uns	L2 set number
7–5	3	line_index	Uns	L2 line index
4–0	5	subblock_index	Uns	L2 subblock index

The PROGRAM_CACHE_FLUSH operations are provided so that the operating system can maintain cache coherence when instructions in program memory are changed. These operations allow program frames to be flushed from the caches, so that subsequent accesses will fetch correct data from program memory.

To flush all entries from a single frame from the L1 and L2 caches, each entry must be flushed with PROGRAM_CACHE_FLUSH. The address must sequence through all values of the L1 set number, and all values of the L2 set number within that page, amounting to 128 flushes. To flush all entries from the L1 and L2 caches, each entry must be flushed with PROGRAM_CACHE_FLUSH_ANY. The address must sequence through all values of the L1 set number, and all values of the L2 set number, comprising a total of 256 flushes. PROGRAM_CACHE_FLUSH_L1 has the same effect on the L1 cache as PROGRAM_CACHE_FLUSH_ANY, without affecting the L2 cache.

Chapter 8: Levels and Protection Domains

8.1 Levels

A stream can execute at one of four privilege levels: LEV_USER. LEV_SUPER (supervisor), LEV_KERNEL, and LEV_IPL (initial program load). Lower levels have fewer privileges. The privilege levels are defined here:

Name	Value	Meaning
Level		
LEV_USER	0	user level
LEV_SUPER	1	supervisor level
LEV_KERNEL	2	kernel level
LEV_IPL	3	initial program load level

User, supervisor, kernel, and IPL level streams are constrained in addressability by the program and data maps. The data map entries define the minimum privilege levels needed to read and to write each segment, and the program map entries define the *exact* privilege level needed to execute from each page.

The LEVEL_ENTER and LEVEL_RTN operations change stream privilege levels. A LEVEL_ENTER must be the first operation executed at an entry point when the caller is from a different privilege level. LEVEL_RTN restores the original privilege level. The current privilege level is expressly *not* directly readable by a stream (although it can be inferred from the program map) to simplify the virtualization of privilege levels.

The domain signal exception is set when the privilege level of the issuing stream is less than the domain signal level in the program state. The domain signal level is increased by the operating system when it finds it necessary to communicate with all streams in its domain, e.g. to prepare for a swap.

8.2 Protection Domains

A processor supports 16 protection domains, each of which implements an address space. Each domain has several registers holding stream resource limits and accounting information. By convention, one of the protection domains is reserved for operating system daemons.

A stream runs in exactly one protection domain, denoted by D. When one stream activates another using the STREAM_CREATE operation, the new stream executes in the same protection domain as its creator and therefore inherits all of its creator's job-context. A stream's protection domain D is read by the privileged DOMAIN_IDENTIFIER_SAVE operation and written by the privileged DOMAIN_LEAVE operation.

Level

Each protection domain has counters controlling stream resource allocation, a data state descriptor and a program state descriptor describing the data and program address spaces. and eight performance counters.

8.3 Stream Resource Control

The seven-bit counter SRES_D contains the total number of streams reserved in the protection domain by STREAM_RESERVE operations. The seven-bit counter SCUR_D maintains a count of the actual number of streams in use. It is constrained by the hardware to be less than or equal to SRES_D, is incremented by the STREAM_CREATE operation, and is decremented by the STREAM_QUIT operation. These counters may also be read by the STREAM_CUR_SAVE and STREAM_RES_SAVE operations.

The seven-bit counter SLIM_D, found in the program state descriptor, contains the maximum number of streams reservable by this protection domain. SLIM_D is an upper bound on SRES_D, the streams currently reserved the protection domain. The operating system sets SLIM_D to prevent the protection domain from monopolizing the available streams.

SLIM_D can actually be set below the current value of SRES_D; because STREAM_QUIT actually decrements SRES_D as well as SCUR_D, both will be coerced lower as streams terminate.

8.4 Data State Descriptor

There is one data state descriptor per protection domain specifying how the data memory operations are interpreted. It is written using the DATA_STATE_RESTORE operation. The descriptor is shown here. Note that the field "retry_limit" is multiplied by four in use. This factor allows retry limits up to 1024. As with memory addresses, the data map limit must have zeros for the six most significant bits in the current implementation.

Bits	Wd	Field Name	Type	Description
DataStateDescriptor				
63–60	4	domain	Uns	the domain to which this descriptor pertains
59–58	2	min_dkill	Level	data minimum level not killed: if a memory operation is selected for issue, and its stream's privilege level is below field "min_dkill", then it fails with result DR_UNIMPLEMENTED_OP, raising data_prot
57–48	10	0		*reserved*
47–28	20	limit	DataSegment	data map limit: the largest data segment number available to the domain; see §6.2
27–20	8	retry_limit	Uns	data memory retry limit: bounds the number of times that a memory operation can be retried before failing and raising the data memory retry exception; see §6.1
19	1	0		*reserved*
18–0	19	base	DataFrame	data map base, added to data segment numbers to yield an offset into local data memory; see §6.2

8.5 Program State Descriptor

There is one program state descriptor per protection domain. It contains several kinds of information relating to instruction interpretation within the domain and is written using the PROGRAM_-STATE_RESTORE operation. This descriptor is defined below:

Program State Descriptor

Bits	Wd	Field Name	Type	Description
ProgramStateDescriptor				
63–60	4	domain	Uns	the domain to which this descriptor pertains
59–58	2	min_pkill	Level	program minimum level not killed: if the privilege level of an issued stream is less than the value in field "min_pkill", then the stream branches to a virtual address set by the scan system, presumably to execute a STREAM_QUIT; after branching, the stream has all traps masked and cannot be pkill'd again
57–56	2	min_psleep	Level	program minimum level not sleeping; if the privilege level of an issued stream is less than the value in field "min_psleep", then all side effects of the instruction are suppressed including counter increments; if both min_pkill and min_psleep are set, psleep has precedence over pkill
ProgramStateDescriptor				
55	1	0		*reserved*
54	1	priv_t0	Boolean	writing target register T0 is privileged, so that unprivileged streams may not freely change the address of the trap handler; see §9.2
53	1	priv_quit	Boolean	STREAM_QUIT operations become privileged, to provide an opportunity to clear the stream's registers before releasing it to the hardware for reallocation; see §2
52–50	3	0		*reserved*
49–48	2	allsig	Level	minimum level not signaled; if a stream is selected for issue and its privilege level is less than the value in field "allsig", then the stream will raise the domain signal exception

ProgramStateDescriptor

47	1	0		*reserved*
46–40	7	slim	Uns	stream limit, SLIM$_D$, which limits the maximum number of streams that may be reserved for this protection domain. see §8.3
39–38	2	0		*reserved*
37–18	20	limit	PageNumber	program map limit, the largest program page number available to the domain: see §7.1
17	1	0		*reserved*
16–0	17	base	ProgFrame	program map base, added to program page numbers to yield an offset into local data memory; see §7.1

Chapter 9: Exceptions and Traps

An exception is an unexpected condition raised by an event in the user program, the operating system, or the hardware. The exception register summarizes exceptional conditions and the result code register describes floating-point and memory exceptions more fully.

Exceptions can cause a trap to be triggered the next time the stream is ready for execution. However, a set exception flag will not trigger a trap if that trap has been disabled by one of the trap-disable bits in the trap mask of the SSW. If an exception is raised while its trap is disabled and the trap is later enabled, the trap will be taken then. Once raised, an exception flag remains set until explicitly cleared by software.

Multiple exceptions can occur simultaneously. For example, if a stream uses lookahead to issue two concurrent loads, the two loads can finish together (between instructions of the issuing stream). Suppose one load raises a data trap 0 exception and the other load raises a data trap 1 exception. It is up to the program (usually a trap handler) to decide the order in which such exceptions are processed.

9.1 Exceptions

The exception register is manipulated using the privileged EXCEPTION_SAVE and EXCEPTION_RESTORE operations. An enumeration for the exceptions is shown below.

Name	Value	Meaning
Exception: Hardware Exceptions		
Ex_Data_HW_Error	57	data memory error or network hardware error
Ex_Prog_HW_Error	56	uncorrectable program memory error
Exception: System Exceptions		
Ex_Instruction_Count	52	instruction count became 0
Ex_Data_Prot	51	data protection
Ex_Prog_Prot	50	program protection
Ex_Poison	49	use of a poisoned register
Exception: Signal Exceptions		
Ex_Domain_Signal	48	domain signal

Exception: User Exceptions		
Ex_Create	44	stream create exception
Ex_Privileged	43	privileged operation
Ex_Data_Alignment	42	unaligned data exception
Ex_Data_Blocked	41	data memory retry exception or data trap
Ex_Float_Extension	40	float software extension exception
Exception: Floating-point Exceptions		
Ex_Float_Invalid	36	float invalid operation
Ex_Float_Zero_Divide	35	float zero divide
Ex_Float_Overflow	34	float overflow
Ex_Float_Underflow	33	float underflow
Ex_Float_Inexact	32	float inexact

Of these exceptions. there are four that are raised by an instruction before it can execute. They are prog_hw_error, prog_prot, poison, and privileged. If one these exceptions is raised while it is masked, the stream will hang. If such an exception is raised and is unmasked, the stream will trap with the trap pc (in T0) pointing to the instruction that caused the exception. If possible, the trap handler could apply some "antidote" and then try to execute the instruction again by returning to T0.

Two exceptions are not raised by the instruction at all: domain_signal and instruction_count. Generally, the trap handler can service the event and return to the program address in T0 to continue. When these exceptions are masked, they are simply ignored.

Most exceptions are caused by the previous instruction. Since that instruction may have contained a jump, its address is lost. Exceptions in this class include create and all the floating-point exceptions.

Presumably, a failing STREAM_CREATE could be handled by reserving a stream and then retrying the create. The easiest way to retry the create is to decrement T0 and jump to it. In this case. there could be no jump in the previous operation since a create is a MAC-operation. A masked create exception will be ignored, although no stream will have been created.

On a floating-point exception, the handler may want to examine the source registers, the operation, and the value written to the destination. Sometimes, it will want to place a new value in the destination based on the above information as well as on some global state. Masked floating-point exceptions are simply ignored.

Finally, there are the data exceptions, which may be caused by any of the last eight instructions executed. These are data_hw_error, data_prot, data_alignment, and data_blocked. To handle these exceptions, the trap routine should check the data result codes and the corresponding DATA_OPA and DATA_OPD state.

The exception register is shown below. There is one bit in the upper half of the exception register for each member of the Exception enumeration shown above. Each of these exception bits can cause a trap, but some SSW bit disables that trap; the left-hand column tells which one.

The lower half of the exception register contains the poison flags. A speculative load operation that would otherwise raise an exception as described in §6.4 instead sets the poison flag corresponding to its destination register. These flags do not trigger traps directly, as confirmed by the "no trap" annotation. Should an instruction use a poisoned register as a source operand, the poison exception will be raised and a trap may then occur. The poisoned register contains a data control descriptor (§6.3), permitting re-execution or diagnosis of the failed load operation by the exception handler.

Trap Disable Bit	Bits	Wd	Field Name	Type	Description
ExceptionRegister: Hardware Exceptions					
	63–58	6	00		*reserved*
hardware	57	1	data_hw_error	Flag	data memory error or network hardware error; see §6.2
hardware	56	1	prog_hw_error	Flag	uncorrectable program memory error; see §7.1
ExceptionRegister: System Exceptions					
	55–53	3	0		*reserved*
system	52	1	instruction_count	Flag	instruction count became 0; see §10
system	51	1	data_prot	Flag	data protection level, map limit, segment limit exceeded, unimplemented op, or unimplemented address; see §6.2
system	50	1	prog_prot	Flag	program protection level, limit violation, or unimplemented address; see §7.1
system	49	1	poison	Flag	use of a poisoned register; see §6.4
ExceptionRegister: Signal Exceptions					
domain signal	48	1	domain_signal	Flag	domain signal: set when the stream level is less than the domain signal level; see §8.1
ExceptionRegister: User Exceptions					
	47–45	3	0		*reserved*
user	44	1	create	Flag	stream create exception: attempt to create more streams than are reserved; see the STREAM_CREATE operation
user	43	1	privileged	Flag	unimplemented or privileged operation; see §3
user	42	1	data_alignment	Flag	unaligned data exception; see §6.1
user	41	1	data_blocked	Flag	data memory retry exception, latency limit exception, or data trap 0 or 1; see §6.1
user	40	1	float_extension	Flag	float software extension; see §5.3

ExceptionRegister: Floating-point Exceptions

	39–37	3	0		*reserved*
float invalid	36	1	float_invalid	Flag	float invalid operation
float zero divide	35	1	float_zero_divide	Flag	float zero divide
float overflow	34	1	float_overflow	Flag	float overflow
float underflow	33	1	float_underflow	Flag	float underflow
float inexact	32	1	float_inexact	Flag	float inexact

ExceptionRegister: Poison Flags

(no trap)	31	1	pf31	Flag	poison flag_{31}
...	...	...			
(no trap)	1	1	pf1	Flag	poison flag_1
	0	1	0		*reserved*

The result code register contains a more detailed description of the results of memory and floating-point operations. The structure of the result code register is described here.

Bits	Wd	Field Name	Type	Description
ResultCode				
ResultCode: A-unit Float Results				
63–56	8	0		*reserved*
55–51	5	A_float_result_reg	Reg	previous A-unit result register
50–48	3	A_float_result_-code	FloatResultCode	A-unit result code
ResultCode: C-unit Float Results				
47–40	8	0		*reserved*
39–35	5	C_float_result_reg	Reg	previous C-unit result register
34-32	3	C_float_result_-code	FloatResultCode	C-unit result code
ResultCode: M-unit Data Results				
31–28	4	dr7	DataResultCode	data result_7
...	...	...		
3–0	4	dr0	DataResultCode	data result_0

The FloatResultCode that is stored in field "A_float_result_code" or field "C_float_result_code" is described below. When no exception or only float_inexact is raised, the result code is set to field "FR_FG". When float_invalid, float_zero_divide, float_overflow, or float_underflow is raised, the result code is set to field "FR_FX". All other result codes are coupled with the float_extension exception. The result register field is written whether or not the result code is nonzero.

FloatResultCode

Name	Value	Meaning
FloatResultCode		
FR_FG	0	float good
FR_IM	3	operand to integer multiply is too large
FR_FX	4	float is exceptional
FR_DZ	5	divide by zero
FR_DR	6	denormalized operand to FLOAT_-RECIP_APPROX
FR_DQ	7	denormalized operand to FLOAT_-RSQRT_APPROX

Each of the four-bit data result fields in the low-order part of the result code register is written when an M-operation completes and contains one of the values shown below.

Name	Value	Meaning
DataResultCode		
DR_NONE	0	the operation completed successfully
DR_DATA_TRAP0	1	data trap 0 exception
DR_DATA_TRAP1	2	data trap 1 exception
DR_DATA_TRAP01	3	both data trap 0 and data trap 1 exception
DR_RETRY_LIMIT	4	data memory retry exception
DR_LATENCY_LIMIT	5	data memory latency exception
DR_DATA_ALIGNMENT	6	data alignment exception
DR_UNIMPLEMENTED_OP	7	unimplemented operation by DATA_-OP_REDO, or aborted by dkill; see §8.4
DR_MAP_LIMIT	8	data map limit exception
DR_PROTECTION_LEVEL	9	data protection level exception
DR_SEGMENT_LIMIT	10	data segment limit exception
DR_UNIMPLEMENTED_-ADDRESS	11	data address unimplemented exception
DR_UNCORRECTABLE_-ERROR	12	uncorrectable data memory exception

The data result fields in the exception register are normally read by the trap handler to diagnose failing memory operations. Data result code i corresponds to the data control descriptor and data retrieved by the DATA_OPA_SAVE or DATA_OPD_SAVE operations with opno i. To simplify trap handling with one or zero failing memory operations, the most recent failing memory operation is relabeled as opno 0.

9.2 Traps

A trap exchanges the ssw.pc with the contents of target register T0 and sets ssw_override mode in ssw. A trap is taken when an exception is raised when its corresponding trap disable bit is clear or a trap disable bit is cleared when a corresponding exception bit is set: see §9.1. A stream that traps does not change its privilege. The trap is lightweight in the sense that only a small amount of state need be saved before control is transferred to a user-supplied exception handler. While the ssw_override flag is set, all traps are masked, lookahead is disabled, and the instruction counter is disabled. In addition, all but synchronizing loads, stores, and int_fetch_adds will retry forever to prevent spurious retries from causing a nested exception. The trap handler should return to the main program using a LEVEL_RTN with the appropriate level. This form of jump will clear ssw_override mode, allowing the next instruction to use the true ssw mode and trap mask bits.

Operations that set T0 are supervisor-privileged if field "priv_t0" is set in the program state descriptor of the protection domain; this option allows auditing of security-relevant events by a trusted (but not necessarily privileged) trap handler. However, restoration of the trap handler entry point when resuming execution of the interrupted activity must be done at privilege level LEV_SUPER or higher if field "priv_t0" is set.

Note that a stream can disable all traps, including system and hardware traps, although it may be unwise to do so. Disabling traps will not necessarily stall the processor or the stream. The operating system can easily regain control by raising field "min_pkill" in the program state descriptor. If the stream's privilege level is less than field "min_pkill", then the stream will branch to a fixed virtual address, generally containing a STREAM_QUIT. If a stream were to encounter a prog_prot, prog_hw_err, or privilege exception after branching in response to pkill, hardware diagnostic intervention is necessary to recover the stream.

There are eight trap registers available for the trap handler to use as temporary storage as it saves or restores processor state. Other uses are discouraged. The trap registers are manipulated by the TRAP_SAVE and TRAP_RESTORE operations. Due to hardware limitations, trap register sets are allocated and deallocated from streams on demand. That is, the first TRAP_RESTORE a stream performs will allocate a trap register set. That set will be deallocated when the stream issues a TRAP_SAVE of TR0. The current implementation provides 32 sets to serve the 128 streams. To protect the operating system, the last trap register set will not be allocated to a user level stream. If a stream tries to allocate a trap register set and fails, that issue is squashed.

Every taken trap counts as a CNT_TRAP event.

Chapter 10: Resource Counters

10.1 Instruction Counter

Each stream has a 16- bit unsigned instruction counter that is intended for debugger support. When an instruction issues, the instruction counter is decremented if the field "count_disable" in the SSW is not set and the counter is not already zero. If the instruction counter becomes zero, then an instruction count exception is raised. The instruction counter is set by the STREAM_COUNT_INST_RESTORE operation and is read by the STREAM_COUNT_INST operation.

10.2 Protection Domain Counters

Each protection domain maintains eight 64-bit resource counters. These counters are only updated every 256 cycles, which limits their resolution.

instruction issue counter
The instruction issue counter increments when an instruction issues in the domain. This counter is read by the COUNT_ISSUES operation.

memory reference counter
The memory reference counter counts the number of memory LOAD, STORE, FETCH_ADD, or STATE operations that are issued in the domain. This counter does not count memory retries or additional memory fetches required for forwarding. When divided by instruction issues, this counter provides an indication of the average number of memory references per instruction. This counter is read by the COUNT_MEMREFS operation.

stream counter
The stream counter is incremented every 256 ticks by the contents of the protection domain's SRES_D counter. When multiplied by 256 and divided by cycles, this counter provides an indication of the average stream usage of the domain. This counter is read by the COUNT_STREAMS operation.

concurrency counter
The concurrency counter is incremented every 256 ticks by the number of memory operations in the protection domain that have issued but not yet completed. When multiplied by 256 and divided by cycles, this counter provides an indication of the average number of memory operations in progress. This counter is read by the COUNT_CONCURRENCY operation.

selectable event counters
The four selectable event counters can be set to count any four of a sizable number of events. The events counted are selected by the event counter select register, which is set by the supervisor-privileged COUNT_SELECT_RESTORE operation and is read by the COUNT_SELECT_SAVE operation. The event counter select register has the structure shown here:

Bits	Wd	Field Name	Type	Description
EventSelect				
63–32	32	00000000		*reserved*
31–24	8	sel_0	CountSource	tag for event counter 0
23–16	8	sel_1	CountSource	tag for event counter 1
15–8	8	sel_2	CountSource	tag for event counter 2
7–0	8	sel_3	CountSource	tag for event counter 3

The value of a selectable event counter is read by the COUNT_EVENTS operation.

The CountSource tag can be one of the values shown below. Setting the tag to an undefined value has undefined results. Setting the tag to denote a dedicated counter has undefined results.

Name	Value	Meaning
CountSource: other operations		
CNT_M_NOP	0	NOP operations executed by the M-unit
CNT_A_NOP	1	NOP operations executed by the A-unit
CNT_C_NOP	2	NOP operations executed by the C-unit
CountSource: target registers		
CNT_TARGET	3	TARGET set operations (not including TARGET_SAVE)
CountSource: data memory		
CNT_LOAD	4	LOAD operations issued
CNT_STORE	5	STORE operations issued
CNT_INT_FETCH_ADD	6	INT_FETCH_ADD operations issued
CNT_MEM_RETRY	7	memory operations retried, including forwarding
CountSource: floating operations		
CNT_FLOAT_ADD	8	FLOAT_ADD and FLOAT_SUB operations
CNT_FLOAT_MUL	9	FLOAT_ADD_MUL operations
CNT_FLOAT_DIV	10	FLOAT_DIV operations
CNT_FLOAT_SQRT	11	FLOAT_SQRT operations
CNT_FLOAT_TOTAL	12	total floating-point operations
CountSource: branches		
CNT_JUMP_EXPECTED	13	expected JUMP or SKIP path taken
CNT_JUMP_UNEXPECTED	14	unexpected JUMP or SKIP path taken
CNT_TRANSFER_TOTAL	15	sum of all transfer operations

CountSource: level switches		
CNT_LEVEL	16	LEVEL_ENTER operations
CountSource: traps		
CNT_TRAP	17	traps taken
CountSource: streams		
CNT_CREATE	18	STREAM_CREATE operations
CNT_QUIT	19	STREAM_QUIT operations
CountSource: dedicated counters		
CNT_ISSUES	128	
CNT_MEMREFS	129	
CNT_STREAMS	130	
CNT_CONCURRENCY	131	

10.3 Processor Counters

Each processor maintains three 64-bit counters.

clock
The clock increments once every tick. It is initialized using the hardware scan mechanism during IPL. By convention, the clocks on all processors are synchronized: that is they agree in value. The clock is read by the CLOCK operation.

phantom counter
The phantom counter counts the number of instruction issue slots unused by its processor. Normally, an instruction will execute to completion once issued, so that true phantoms are the main source of unused slots. However, some exceptions, such as a poison exception, will cause the triggering instruction to be aborted, "wasting" an issue slot. These aborted instructions count as phantoms as well.

ready counter
The ready counter on each processor sums the total number of streams ready at each tick of its processor. However, a stream that issues as soon as it becomes ready will not contribute to this counter. Due to stream scheduling constraints, a ready stream may wait a few cycles before it issues, even on a processor with free issue slots.

Chapter 11: Operation Descriptions

11.1 Notation

A TERA MTA assembly language program has the same syntactic form as a sequence of Lisp expressions. Each instruction has the format:

(INST *lookahead M-operation A-operation C-operation*)

If any of the M-, A-, or C-operations is not specified, then the assembler will fill the missing operation with a NOP. Operations should be specified in order when one of them can be executed by more than one functional unit, e.g. FLOAT_ADD.

An operation belongs to exactly one *group*. The operations in a group have minor differences, such as in the way operands are addressed, whether the M-, A-, or C-unit does the operation, and so forth. Each page in this section describes an operation group. The following page describes a sample operation group.

(OPERATION_1 *operand-template*) $\quad {}_{64}\ldots{}_{42}u{}_{37}v{}_{32}01{}_{27}02{}_{21}\ldots{}_{0}$ CLASS

pseudo code description of operation_1
{**where** *variable* $\in$ *set*}

(OPERATION_2 *operand-template*) $\quad {}_{64}\ldots{}_{42}u{}_{37}v{}_{32}01{}_{27}03{}_{21}\ldots{}_{0}$ CLASS

pseudo code description of operation_2
{**where** *variable* $\in$ *set*}

The operations in this group (here, OPERATION_1 and OPERATION_2) are described in more detail in this section.

The CLASS describes the set of M, A, and C instruction fields used by this operation.

An identifier always refers to the same value within the description of an operation. The identifiers $r, s, t, u, v, w, x, y, z$ always refer to the contents of a register in a fixed position within the 64-bit instruction. Subscripts on identifiers are bit subscripts. Unless otherwise stated, the range of a subscript is from 63 down to 0. Bit numbers increase from right to left. An identifier denoting an immediate constant is quoted, e.g. *'disp*. The range of an immediate constant is always constrained by a **where** clause.

The clause "{**where** *predicate*}" is a *constraint*. The *predicate* is a Boolean function that must be true. The most common constraints bound the range of immediate constants.

The pseudo code description uses a conventional Algol-like language containing flow statements, assignment statements, and expressions, built from operators and operands. Operators are described below. An operand may be a constant (interpreted in base 10); a quoted identifier, such as *'disp*, denoting an immediate value; or a non-quoted identifier, such as r, denoting the contents of the register addressed by the value bound to that identifier.

The assembly code prototype for an operation is: "(OPERATION *operand-template*)". The encoding for each operation is described using *fields*. The fields used by an operation are listed from left to right within the word, from bit 63 to bit 0. Each field has two parts. The field *end* is the field's low-order bit number. The field *fill* can be an identifier, which stands for its value; a literal constant, which is represented in hexadecimal; or an ellipses representing "holes" in the object encoding. For example, the encoding for OPERATION_1 denotes a field starting at bit 41 and ending at bit 37 containing the register number u, a field starting at bit 36 and ending at bit 32, containing the register number v, a field starting at bit 31 and ending at bit 27 containing the literal value "01_{16}", and finally a field starting at bit 26 and ending at bit 21 containing the literal value "02_{16}".

For the most part, the order of fields in the object code encoding is the same as the order of fields in assembly language. Deviations from this rule are noted explicitly. In any event, the mapping from assembly to machine code is manifest in the encodings.

EXAMPLES

An example of how the instruction might be used is given here.

RAISES

The exceptions that the operation may raise are enumerated here.

COUNTS AS

The event counters that are incremented by this operation.

SEE ALSO

The names of related operations and section numbers are given here

11.2 Operation Naming Conventions

The mnemonics for operations are chosen according to these general rules.

- The mnemonics for most operations start with the name of the data type being manipulated. The principle exceptions are the STORE family, the SHIFT family, the JUMP family, and the miscellaneous operations dealing with program state. The data type prefixes are shown in Figure 11.1.
- Mnemonics ending with "_TEST" generate a condition code.
- Mnemonics containing "_RESTORE" move value(s) from general purpose registers into special registers.
- Mnemonics containing "_SAVE" move value(s) from special registers into general purpose registers.
- Mnemonics containing "_IMM" contain a small immediate constant operand. Some of these operations accept one value in the assembly code, but place another value in the object code. For example, INT_ADD_IMM takes and adds a value from 1 to 32, but the value put in the object code ranges from 0 to 31; the hardware increments this value to produce the desired sum.
- Mnemonics containing "_MAP" deal with the program or data map.

data type	what
BIT	a word of bits
BIT_MAT	an 8 * 8 matrix of bits packed into a word
COUNT	a resource count
DATA	M-unit state
DOMAIN	a protection domain
EXCEPTION	an exception
FLOAT	a floating-point number
INT	a signed integer
LEVEL	a privilege level
LOGICAL	a 64-bit wide logical value: 0(false), 1(true) or -1(true).
PTR	a pointer to memory, with access control
STATE	memory access state bits
STREAM	an instruction stream
UNS	an unsigned integer

FIGURE 11.1: Data Type Prefixes

- Mnemonics containing "_AC" use access control from the operation when computing an address.
- Mnemonics containing "_INDEX" use scaled indexing when computing an address.
- Mnemonics containing "_DISP" use scaled displacements when computing an address.

11.3 Pseudo-code Operators

Infix operators in expressions have the precedence and associativity customary to the C language. Parentheses are used in complex expressions to avoid confusion. The operator ";" is lowest precedence and separates sequentially executed statements[1]. In addition, the operators shown in Figure 11.2 are used.

[1]Note that in ISP ";" means parallel execution of the statements. We adopt the conventional Algol semantics.

$\leftarrow$	assignment
store	store to memory
load	load from memory
$\vee$	logical or
$\wedge$	logical and
$\oplus$	logical exclusive-or
$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division
min	minimum value
max	maximum value
$\sqrt{}$	square root
tally	the number of 1 bits
$\in$	is a member of
$\gg_a$	shift right, with sign bit filling
$\gg$	shift right, with 0 filling
$\ll$	shift left, with 0 filling
$\hookrightarrow$	rotate right
$\hookleftarrow$	rotate left
$[i \ldots j]$	range $i, i+1, \ldots, j-1, j$
a_i	bit number i from a
a_r	bits in the range r from a

FIGURE 11.2: Pseudo-code Operators

(BIT_AND $t\ u\ v$) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}01_{27}0C_{21}\cdots{}_{0}$ A

$t \leftarrow u \wedge v$

(BIT_AND $x\ y\ z$) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}z_{6}12_{0}$ C

$x \leftarrow y \wedge z$

(BIT_AND_TEST $t\ u\ v$) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}01_{27}0D_{21}\cdots{}_{0}$ A

$t \leftarrow u \wedge v$

(BIT_AND_TEST $x\ y\ z$) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}z_{6}13_{0}$ C

$x \leftarrow y \wedge z$

These operations compute bitwise and.

BIT_AND_TEST never generates overflow/NaN or carry.

RAISES

(nothing)

BIT_AND_

(BIT_IMP t u v) $\quad {}_{64}\ldots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,07\,{}_{27}\,0C\,{}_{21}\ldots{}_{0}$ A

$t \leftarrow \neg u \vee v$

(BIT_IMP_TEST t u v) $\quad {}_{64}\ldots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,07\,{}_{27}\,0D\,{}_{21}\ldots{}_{0}$ A

$t \leftarrow \neg u \vee v$

These operations compute bitwise implication.

BIT_IMP_TEST never generates overflow/NaN or carry.

RAISES

(nothing)

(BIT_LEFT_ONES x y) $_{64}\cdots{}_{21}x_{16}y_{11}04_{6}00_{0}$ C

$x \leftarrow \min\{i|(y \ll i) \geq 0\}$

(BIT_LEFT_ONES_TEST x y) $_{64}\cdots{}_{21}x_{16}y_{11}04_{6}01_{0}$ C

$x \leftarrow \min\{i|(y \ll i) \geq 0\}$

(BIT_LEFT_ZEROS x y) $_{64}\cdots{}_{21}x_{16}y_{11}05_{6}00_{0}$ C

$x \leftarrow 64 - \min\{i|(y \gg i) = 0\}$

(BIT_LEFT_ZEROS_TEST x y) $_{64}\cdots{}_{21}x_{16}y_{11}05_{6}01_{0}$ C

$x \leftarrow 64 - \min\{i|(y \gg i) = 0\}$

These operations respectively return the number of consecutive 1- or 0-bits on the left end of the word in y.

The _TEST versions of these operations generate carry when the result is 64 and never generate overflow/NaN.

EXAMPLES

A linear search for the leftmost 0-bit in a contiguous block of words pointed to by p could be done using the loop shown below. The code returns the bit offset from the leftmost bit of the vector, and assumes that a zero will eventually be found.

```
             (INST 0  (LOAD n p) (TARGET_DISP t_loop loop) (REG_MOVE bn r0))
(LABEL loop)
             (INST 7  (INT_ADD_IMM p p 8) (BIT_LEFT_ONES_TEST b n))
             (INST 0  (LOAD n p)
                      (INT_ADD bn bn b)
                      (JUMP IF_C c0 t_loop))
```

RAISES

(nothing)

SEE ALSO

BIT_RIGHT_

BIT_LEFT_

(BIT_MASK *t top bot*) $\quad {}_{64}\cdots{}_{47}t_{42}3_{39}top_{33}bot_{27}00_{21}\cdots{}_{0}$ **A**

```
    for $i \in [0 \ldots 63] : t_i — ( 'top \geq i ) \mp ( i \geq 'bot ) \ominus ( 'top \geq 'bot )$
        {where $'top \in [0 \ldots 63]. \ 'bot \in [0 \ldots 63]$}
```

A mask is generated that contains 1-bits from bit positions [*bot* ... *top*] and 0-bits elsewhere. If *top* is less than *bot* then the bit positions set to 1 are [0 ... *top*] and [*bot* ... 63], generating a complement mask.

EXAMPLES

A mask containing ones in bit positions [21 ... 46] and zeros elsewhere is generated by (BIT_MASK t 46 21). Its complement is generated by (BIT_MASK t 20 47).

RAISES

(nothing)

SEE ALSO

INT_IMM

(BIT_MAT_OR t u v)

```
   ... t  u  v  10  0C  ...        A
64   47 42 37 32  27  21   0
```

for $i,j \in [0\ldots7] : t_{8*i+j} \leftarrow \bigvee_{k=0}^{7}(u_{8*i+k} \wedge v_{8*k+j})$

(BIT_MAT_TRANSPOSE t u)

```
   ... t  u  1B  00  08  ...       A
64   47 42 37  32  27  21   0
```

for $i,j \in [0\ldots7] : t_{8*i+j} \leftarrow u_{8*j+i}$

(BIT_MAT_XOR t u v)

```
   ... t  u  v  11  0C  ...        A
64   47 42 37 32  27  21   0
```

for $i,j \in [0\ldots7] : t_{8*i+j} \leftarrow \bigoplus_{k=0}^{7}(u_{8*i+k} \wedge v_{8*k+j})$

These operations provide the basic support for multiply and transpose of bit matrices. Each byte of a word represents a row of an 8×8 matrix. Matrices of arbitrary size can be represented as matrices of 8×8 blocks.

EXAMPLES

The word 8040201008040201_{16} is the identity matrix. For either BIT_MAT_XOR or BIT_MAT_OR the matrix 0102040810204080_{16} in u will reverse the bytes in v, leaving the bit order unchanged. The same matrix in v will reverse the bits in each byte of u, leaving the byte order unchanged.

RAISES

(nothing)

SEE ALSO

BIT_OR, BIT_AND, BIT_XOR

(BIT_MERGE $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}26_{21}\cdots{}_{0}$ A

for $i \in [0 \ldots 63] : t_i \leftarrow$ **if** w_i **then** u_i **else** v_i

(BIT_MERGE_TEST $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}27_{21}\cdots{}_{0}$ A

for $i \in [0 \ldots 63] : t_i \leftarrow$ **if** w_i **then** u_i **else** v_i

These operations select a bit from u if the corresponding bit in w is set; otherwise the bit from v is selected.

BIT_MERGE_TEST never generates overflow/NaN or carry.

RAISES

(nothing)

(BIT_NAND $t\ u\ v$) ${}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}\mathrm{06}_{27}\mathrm{0C}_{21}\cdots{}_{0}$ **A**

$t — \neg(u \wedge v)$

(BIT_NAND_TEST $t\ u\ v$) ${}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}\mathrm{06}_{27}\mathrm{0D}_{21}\cdots{}_{0}$ **A**

$t — \neg(u \wedge v)$

These operations compute bitwise negated and.

BIT_NAND_TEST never generates overflow/NaN or carry.

RAISES

(nothing)

BIT_NAND_

(BIT_NIMP t u v) $\quad {}_{64}\cdots{}_{47}t{}_{42}u{}_{37}v{}_{32}\mathrm{00}{}_{27}\mathrm{0C}{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow u \wedge \neg v$

(BIT_NIMP x y z) $\quad {}_{64}\cdots{}_{21}x{}_{16}y{}_{11}z{}_{6}\mathrm{10}{}_{0}$ **C**

$x \leftarrow y \wedge \neg z$

(BIT_NIMP_TEST t u v) $\quad {}_{64}\cdots{}_{47}t{}_{42}u{}_{37}v{}_{32}\mathrm{00}{}_{27}\mathrm{0D}{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow u \wedge \neg v$

(BIT_NIMP_TEST x y z) $\quad {}_{64}\cdots{}_{21}x{}_{16}y{}_{11}z{}_{6}\mathrm{11}{}_{0}$ **C**

$x \leftarrow y \wedge \neg z$

These operations compute bitwise negated implication.

BIT_NIMP_TEST never generates overflow/NaN or carry.

RAISES

(nothing)

(BIT_NOR t u v) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}04_{27}0C_{21}\cdots{}_{0}$ A

$t \leftarrow \neg(u \vee v)$

(BIT_NOR_TEST t u v) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}04_{27}0D_{21}\cdots{}_{0}$ A

$t \leftarrow \neg(u \vee v)$

These operations compute bitwise negated or.

BIT_NOR_TEST never generates overflow/NaN or carry.

RAISES

(nothing)

BIT_NOR_

(BIT_ODD_AND t u v) ${}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathrm{09}_{27}\mathrm{0C}_{21}\ldots{}_{0}$ A

$t \leftarrow (-1) * \bigoplus_{j=0}^{63} u_j \wedge v_j$

(BIT_ODD_AND_TEST t u v) ${}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathrm{09}_{27}\mathrm{0D}_{21}\ldots{}_{0}$ A

$t \leftarrow (-1) * \bigoplus_{j=0}^{63} u_j \wedge v_j$

(BIT_ODD_NIMP t u v) ${}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathrm{08}_{27}\mathrm{0C}_{21}\ldots{}_{0}$ A

$t \leftarrow (-1) * \bigoplus_{j=0}^{63} u_j \wedge \neg v_j$

(BIT_ODD_NIMP_TEST t u v) ${}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathrm{08}_{27}\mathrm{0D}_{21}\ldots{}_{0}$ A

$t \leftarrow (-1) * \bigoplus_{j=0}^{63} u_j \wedge \neg v_j$

(BIT_ODD_OR t u v) ${}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathrm{0B}_{27}\mathrm{0C}_{21}\ldots{}_{0}$ A

$t \leftarrow (-1) * \bigoplus_{j=0}^{63} u_j \vee v_j$

(BIT_ODD_OR_TEST t u v) ${}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathrm{0B}_{27}\mathrm{0D}_{21}\ldots{}_{0}$ A

$t \leftarrow (-1) * \bigoplus_{j=0}^{63} u_j \vee v_j$

(BIT_ODD_XOR t u v) ${}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathrm{0A}_{27}\mathrm{0C}_{21}\ldots{}_{0}$ A

$t \leftarrow (-1) * \bigoplus_{j=0}^{63} u_j \ominus v_j$

(BIT_ODD_XOR_TEST t u v) ${}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathrm{0A}_{27}\mathrm{0D}_{21}\ldots{}_{0}$ A

$t \leftarrow (-1) * \bigoplus_{j=0}^{63} u_j \ominus v_j$

These operations do a two-operand bitwise operation and compute the parity of the result. The value stored in t is either all 1's or all 0's.

RAISES

(nothing)

(BIT_OR *t u v*) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}03_{27}0C_{21}\cdots{}_{0}$ A

$t \leftarrow u \vee v$

(BIT_OR *x y z*) $_{64}\cdots{}_{21}x_{16}y_{11}z_{6}16_{0}$ C

$x \leftarrow y \vee z$

(BIT_OR_TEST *t u v*) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}03_{27}0D_{21}\cdots{}_{0}$ A

$t \leftarrow u \vee v$

(BIT_OR_TEST *x y z*) $_{64}\cdots{}_{21}x_{16}y_{11}z_{6}17_{0}$ C

$x \leftarrow y \vee z$

These operations compute bitwise or.

BIT_OR_TEST never generates overflow/NaN or carry.

RAISES

(nothing)

BIT_OR_

(BIT_PACK t u v) $_{64}\ldots_{47}\ t\ _{42}\ u\ _{37}\ v\ _{22}\ 14\ _{27}\ \mathrm{OC}\ _{21}\ldots_{0}$ A

for $i \in [0 \ldots 63] : t_i \leftarrow \exists j\{\neg u_j \wedge v_j \wedge (\text{tally}\ (\neg[u_j \ldots u_0]) = (i - 1))\}$

These operations are used for packing bit fields from v into t under control of the mask found in u.

Bits from v selected by zeros in u are packed consecutively into bit positions in the destination register t, starting from the right. Unfilled positions in t are set to zero.

EXAMPLES

This example only shows 8 bits, and describes the operations for BIT_PACK. Capital letters stand for arbitrary bit values.

BIT_PACK	register	what
11011001	u	source selector mask
RSTVWXYZ	v	source
00000TXY	t	destination

Bits T, X and Y are selected by the source mask, and are written into the destination register t in that order, packed to the right, with zeros filled in on the left.

RAISES

(nothing)

SEE ALSO

BIT_UNPACK_1

(BIT_RIGHT_ONES x y) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}06_{6}00_{0}$ $\quad$ C

$x \leftarrow \min\{i|(y \gg i) \text{ is even}\}$

(BIT_RIGHT_ONES_TEST x y) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}06_{6}01_{0}$ $\quad$ C

$x \leftarrow \min\{i|(y \gg i) \text{ is even}\}$

(BIT_RIGHT_ZEROS x y) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}07_{6}00_{0}$ $\quad$ C

$x \leftarrow 64 - \min\{i|(y \ll i) = 0\}$

(BIT_RIGHT_ZEROS_TEST x y) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}07_{6}01_{0}$ $\quad$ C

$x \leftarrow 64 - \min\{i|(y \ll i) = 0\}$

These operations respectively return the number of consecutive 1- or 0-bits on the right end of the word in y.

The _TEST versions of these operations generate carry when the result is 64 and never generate overflow/NaN.

RAISES

(nothing)

SEE ALSO

BIT_LEFT_

BIT_RIGHT_

(BIT_TALLY t u) $\quad {}_{64}\ldots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,\mathtt{00}\,{}_{32}\,\mathtt{0F}\,{}_{27}\,\mathtt{0C}\,{}_{21}\ldots{}_{0}$ A

$t - \sum_{j=0}^{63} u_j$

(BIT_TALLY_TEST t u) $\quad {}_{64}\ldots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,\mathtt{00}\,{}_{32}\,\mathtt{0F}\,{}_{27}\,\mathtt{0D}\,{}_{21}\ldots{}_{0}$ A

$t - \sum_{j=0}^{63} u_j$

These operations count the number of 1-bits in the u register value. The _TEST versions never generate overflow/NaN and generate carry when $t = 1$.

RAISES

(nothing)

(BIT_UNPACK_1 t u v) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}15_{27}0C_{21}\cdots{}_{0}$ A

```
for $i \in [0 \ldots 63]$ {
    $shift_i \leftarrow i + 1 - \text{tally}(\neg[u_i \ldots u_0])$
    $s_i \leftarrow shift_i \wedge 63$
    $d_i \leftarrow shift_i \wedge 15$
    $t_{i-d_i} \leftarrow v_{i-s_i}$
}
```

(BIT_UNPACK_2 t u v) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}16_{27}0C_{21}\cdots{}_{0}$ A

```
for $i \in [0 \ldots 63]$ {
    $shift_i \leftarrow i + 1 - \text{tally}(\neg[u_i \ldots u_0])$
    $s_i \leftarrow shift_i \wedge 15$
    $d_i \leftarrow shift_i \wedge 3$
    $t_{i-d_i} \leftarrow v_{i-s_i}$
}
```

(BIT_UNPACK_3 t u v) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}17_{27}0C_{21}\cdots{}_{0}$ A

```
for $i \in [0 \ldots 63]$ {
    $shift_i \leftarrow i + 1 - \text{tally}(\neg[u_i \ldots u_0])$
    $s_i \leftarrow shift_i \wedge 3$
    $d_i \leftarrow shift_i \wedge 0$
    $t_{i-d_i} \leftarrow \neg u_i \wedge v_{i-s_i}$
}
```

The BIT_UNPACK_1, BIT_UNPACK_2, BIT_UNPACK_3 operation sequence is used for unpacking bit data in v into result t under control of the mask found in u.

Using a fixed mask and the result of BIT_UNPACK_1 as the input data for BIT_UNPACK_2 (and BIT_UNPACK_2 for BIT_UNPACK_3), bits from v are packed consecutively into bit positions in the destination register t selected by zeros in u. Unselected positions in t are set to zero. Extra bits from v are discarded. The operation packs from right to left.

EXAMPLES

This example only shows 8 bits, and describes the operations for the three-operation bit unpack sequence. Capital letters stand for arbitrary bit values.

bit unpack	register	what
01001110	u	destination selector mask
RSTVWXYZ	v	source
W0XY000Z	t	destination

Bits W, X, Y and Z are written into the destination register t in that order, but in positions selected by the mask in u.

RAISES

(nothing)

SEE ALSO

BIT_PACK

BIT_UNPACK_

(BIT_XNOR t u v) $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,\mathtt{05}\,{}_{27}\,\mathtt{0C}\,{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow \neg(u \oplus v)$

(BIT_XNOR_TEST t u v) $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,\mathtt{05}\,{}_{27}\,\mathtt{0D}\,{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow \neg(u \oplus v)$

These operations compute bitwise negated exclusive-or.

BIT_XNOR_TEST never generates overflow/NaN or carry.

RAISES

(nothing)

(BIT_XOR t u v) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}02_{27}0C_{21}\cdots{}_{0}$ A

$t \leftarrow u \oplus v$

(BIT_XOR x y z) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}z_{6}14_{0}$ C

$x \leftarrow y \oplus z$

(BIT_XOR_TEST t u v) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}02_{27}0D_{21}\cdots{}_{0}$ A

$t \leftarrow u \oplus v$

(BIT_XOR_TEST x y z) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}z_{6}15_{0}$ C

$x \leftarrow y \oplus z$

These operations compute bitwise exclusive-or.

BIT_XOR_TEST never generates overflow/NaN or carry.

RAISES

(nothing)

BIT_XOR_

(BREAK) $_{64}\ldots_{47}\,00\,_{42}\,01\,_{37}\,00\,_{32}\,00\,_{27}\,00\,_{22}\ldots_{0}$ A

Raise privileged operation exception

This operation raises a privileged operation exception. It may be used by the debugger to implement breakpoints.

RAISES

privileged

(CLOCK x y) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}19_{6}00_{0}$ C

x — clock — y

This operation returns the contents of the 64-bit clock register. which increments by one on each clock tick. Normally. the clock register contents are synchronized across all processors in a system.

RAISES

(nothing)

SEE ALSO

§10.3

(COUNT_CONCURRENCY *t*) ${}_{64}\cdots{}_{47}t_{42}0_{37}1B_{32}13_{27}08_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ (concurrency counter)$_D$

(COUNT_EVENTS *t ec*) ${}_{64}\cdots{}_{47}t_{42}0_{37}1B_{32}5_{29}ec_{27}08_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ (event counter at ec)$_D$

(COUNT_ISSUES *t*) ${}_{64}\cdots{}_{47}t_{42}0_{37}1B_{32}10_{27}08_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ (instruction issue counter)$_D$

(COUNT_MEMREFS *t*) ${}_{64}\cdots{}_{47}t_{42}0_{37}1B_{32}11_{27}08_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ (memory reference counter)$_D$

(COUNT_STREAMS *t*) ${}_{64}\cdots{}_{47}t_{42}0_{37}1B_{32}12_{27}08_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ (stream counter)$_D$

These operations read one of the eight counters in the protection domain D of the executing stream. The four event counters each can be set independently to one of the count sources by the supervisor-privileged COUNT_SELECT_RESTORE operation. Each event counter has an eight-bit CountSource tag that determines what is to be counted. These tags are packed in the count_select register. A description of the counters and the encoding of the CountSource tag is described in §10.2.

RAISES

(nothing)

SEE ALSO

COUNT_SELECT_RESTORE, CLOCK, STREAM_COUNT_INST

(COUNT_SELECT_RESTORE *u*) $_{64}\cdots{}_{47}0_{44}0_{42}u_{37}00_{32}10_{27}0A_{21}\cdots{}_{0}$ **A**

(event counter select)$_D$ — *u*

(COUNT_SELECT_SAVE *t*) $_{64}\cdots{}_{47}t_{42}00_{37}1B_{32}1C_{27}08_{21}\cdots{}_{0}$ **A**

t — (event counter select)$_D$

These operations allow the event counter select register to be read and written. The COUNT_-SELECT_RESTORE operation requires supervisor privilege. This register contains the four select tags for the programmable event counters, described in §10.2.

RAISES

privileged

SEE ALSO

COUNT_

COUNT_SELECT_

(COUNT_PHANTOMS *t*) $_{64}\cdots{}_{47}\,t\,_{42}\,0\,_{37}\,1B\,_{32}\,18\,_{27}\,08\,_{21}\cdots{}_{0}$ **A**

t — (processor phantom counter)

(COUNT_READY *t*) $_{64}\cdots{}_{47}\,t\,_{42}\,0\,_{37}\,1B\,_{32}\,19\,_{27}\,08\,_{21}\cdots{}_{0}$ **A**

t — (processor ready counter)

These operations read the processor's phantom and ready counters. The phantom counter sums the number of ticks where no stream was ready, so that the processor utilization may be measured. The ready counter sums the number of ready but not issuing streams at each tick, so that the average ready pool size and average waiting time may be measured.

RAISES

(nothing)

SEE ALSO

CLOCK

(DATA_MAP_FLUSH *s*) $_{64}\cdots{}_{61}0_{56}s_{51}F_{47}\cdots{}_{21}00_{16}00_{11}02_{6}02_{0}$ MC

flush (data map at *s*) from data map cache

(DATA_MAP_FLUSH_ANY *s*) $_{64}\cdots{}_{61}0_{56}s_{51}F_{47}\cdots{}_{21}00_{16}00_{11}03_{6}02_{0}$ MC

flush any (data map at *s*) from data map cache

These are supervisor-privileged operations to maintain consistency in the data address translation cache.

The domain in Bits 63–60 and segment in Bits 41–28 of *s* address the data map entry; other bits of *s* are ignored. A violation of the map limit will not raise a data map limit exception. The DATA_MAP_FLUSH operation is used to flush a single map entry, as after changing the data map in data memory. The DATA_MAP_FLUSH_ANY operation is used to flush any map entry for the specified domain from the cache. Since the cache is not fully associative, up to 512 flushes may be required—each flush should specify a different segment modulo 512. See §6.2.

These operations are subject to the min_dkill level (see §8.4) based on the domain. Therefore, it may be necessary to perform a DATA_STATE_RESTORE on the specified domain prior to flushing.

RAISES

data_prot, privileged

SEE ALSO

PROGRAM_MAP_

DATA_MAP_

(DATA_OPA_SAVE *r opno*) $_{64}\cdots{}_{61}\,r\,_{56}\,00\,_{51}\,\mathrm{F}\,_{47}\cdots{}_{21}\,00\,_{16}\,00\,_{11}\,0\,_{9}\,opno\,_{6}\,04\,_{0}$ MC

r — address state of operation *opno*

(DATA_OPD_SAVE *r opno*) $_{64}\cdots{}_{61}\,r\,_{56}\,00\,_{51}\,\mathrm{F}\,_{47}\cdots{}_{21}\,00\,_{16}\,00\,_{11}\,1\,_{9}\,opno\,_{6}\,04\,_{0}$ MC

r — data state of operation *opno*

(DATA_OP_REDO *r s*) $_{64}\cdots{}_{61}\,r\,_{56}\,s\,_{51}\,\mathrm{F}\,_{47}\cdots{}_{21}\,00\,_{16}\,00\,_{11}\,10\,_{6}\,04\,_{0}$ MC

perform memory operation in *s* with data to or from *r*

These operations save and (re)execute failed memory references. They are normally used by trap handlers. A stream may have eight memory references pending in the M-unit. Each reference is described by two words. One word contains the data value (if any), and the other contains address and control information. The data value must be read after the address word. The eight data result fields of the result code register describe which of these memory references are exceptional after a trap.

DATA_OPA_SAVE retrieves a Data Control Descriptor containing address and control information from the M-unit for the operation denoted by *opno* and places that information into register *r* (see §6.3). To allow an *opno* of zero to select the descriptor which must be saved first, the lookahead index is added to *opno* modulo eight. DATA_OPD_SAVE retrieves the corresponding data from the M-unit for the operation denoted by *opno* and places that information into register *r*. Note that DATA_OPA_SAVE must be performed before DATA_OPD_SAVE for each *opno*.

The DATA_OP_REDO operation re-executes an M-unit operation, given a Data Control Descriptor in register *s* and the corresponding data in register *r*. The "original" register number in field "dest_-reg" of *s* is ignored; a load operation will place its result in register *r*. DATA_OP_REDO raises data memory exceptions (and otherwise behaves) just as the "original" operation, as described by *r* and *s*, would have. If the value in *s* does not indicate one of the defined operations for the Data Control Descriptor, a data_prot exception is raised, and the result code is set to DR_UNIMPLEMENTED_-OP.

Since the value of *s* must be interpreted before it is known whether the DATA_OP_REDO operation will actually read or write *r*, poison is checked for reading *r*.

RAISES

data_prot, data_alignment, data_blocked

SEE ALSO

EXCEPTION_, RESULTCODE_, §6.3

(**DATA_STATE_RESTORE** *s*)

```
 ...  0  s  F  ...  00  00  04  02        MC
64  61 56 51 47   21  16  11   6   0
```

(data state descriptor for domain in *s*) — *s*

This supervisor-privileged operation is used to set the data state descriptor.

The value *s* is the data state descriptor: see §8.4. The Bits 63–60 of *s* specify the protection domain. i.e. the data state descriptor is self-tagged.

RAISES

privileged

SEE ALSO

PROGRAM_STATE_

DATA_STATE_

```
(DOMAIN_ENTER)                              ...0  0  00   00   0D   0A  ...     A
                                          64    47 44 42   37   32   27   21    0
    if limbo > 0 then
        limbo — limbo - 1;
        SRES_D  — SRES_D + 1;
        SCUR_D  — SCUR_D + 1;
    else
        raise create exception
    end

(DOMAIN_LEAVE u)                            ...0  0  u  00   0C   0A  ...      A
                                          64    47 44 42 37   32   27   21    0
    limbo — limbo + 1;
    SCUR_D  — SCUR_D - 1;
    SRES_D  — SRES_D - 1;
    D  — u;
    if limbo >= 128 then
        raise create exception
    end
```

These are a supervisor-privileged operations to change protection domains. No lookahead is allowed across a DOMAIN_LEAVE/DOMAIN_ENTER pair.

Executing a DOMAIN_LEAVE, DOMAIN_ENTER sequence will change the domain to the specified value. The protection domain D of the stream changes to that specified in Bits 3–0 of register *u*. The current pc address in the SSW must map to the same page in both the new and old domains.

The create exception protects against a DOMAIN_ENTER without a matching DOMAIN_LEAVE, or too many domain changes in progress at once.

RAISES

privileged, create

SEE ALSO

LEVEL_RTN

(DOMAIN_IDENTIFIER_SAVE *t*)

```
...  t  08  00  00  00  ...        A
64    47 42   37  32  27  21    0
```

t — D. the current protection domain identifier

This is a supervisor-privileged operation used to retrieve the identity of the current protection domain D.

RAISES

privileged

DOMAIN_ID_

(EXCEPTION_RESTORE *u*)

```
 . . . 0  0  u  00  09  0A . . .        A
64   47 44 42 37  32  27  21    0
```

EXCEPTION — *u*

(EXCEPTION_SAVE *x*)

```
 . . . x  00  1C 00        C
64   21 16  11  6  0
```

x — EXCEPTION

These operations manipulate the exception register, which contains the exception bits that record unusual and possibly significant events resulting from instruction execution. Every exception bit causes a trap unless that trap is disabled by the appropriate bit in the trap mask of the SSW.

Additional information about the most recent floating-point and memory exceptions are found in the result code register.

RAISES

(nothing)

SEE ALSO

RESULTCODE_, §9.1

(FLOAT_CEIL *t u*) $_{64}\cdots{}_{47}t_{42}u_{37}18_{32}03_{27}08_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ (float ceiling of float u) $* 2^{-1074}$

(FLOAT_CHOP *t u*) $_{64}\cdots{}_{47}t_{42}u_{37}18_{32}01_{27}08_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ (float integer chop of float u) $* 2^{-1074}$

(FLOAT_FLOOR *t u*) $_{64}\cdots{}_{47}t_{42}u_{37}18_{32}02_{27}08_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ (float floor of float u) $* 2^{-1074}$

(FLOAT_NEAR *t u*) $_{64}\cdots{}_{47}t_{42}u_{37}18_{32}00_{27}08_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ (float integer nearest float u) $* 2^{-1074}$

(FLOAT_ROUND *t u*) $_{64}\cdots{}_{47}t_{42}u_{37}18_{32}0C_{27}08_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ (float integer round of float u) $* 2^{-1074}$

These operations scale 64-bit floating-point numbers by 2^{-1074}, then round the result into floating-point numbers. Since the scaling reduces 1.0 to the minimum denormalized number, the effect is to round the argument to an integer. The roundings are directed as in IEEE Standard 754. FLOAT_ROUND uses the rounding mode in the SSW.

Note that the float_inexact exception is never raised, since it is expected for these operations.

RAISES

(nothing)

SEE ALSO

INT_, UNS_

FLOAT_

(FLOAT_ADD t u v) $_{64}\cdots{}_{47}\,t\,_{42}\,u\,_{37}\,v\,_{32}\,10\,_{27}\,\mathrm{0E}\,_{21}\cdots{}_{0}$ A

$t \leftarrow u + v$, floating point

(FLOAT_ADD x y z) $_{64}\cdots{}_{21}\,x\,_{16}\,y\,_{11}\,z\,_{6}\,\mathrm{0C}\,_{0}$ C

$x \leftarrow y + z$, floating point

This operation computes the floating-point sum of two numbers.

RAISES

float_invalid, float_overflow, float_inexact

COUNTS AS

CNT_FLOAT_ADD, CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_ADD_MUL

(FLOAT_ADD_MUL t u v w) $_{64} \cdots _{47} t _{42} u _{37} v _{32} w _{27} 30 _{21} \cdots _{0}$ **A**

$t \leftarrow u + v * w$, floating point

This operation does a floating-point multiply followed by a floating-point add, counting as two floating-point operations. If u is register 0, only a floating-point multiply is performed (to preserve the sign of a zero result).

Only one rounding operation is performed, enhancing accuracy and greatly facilitating doubled precision operations.

RAISES

float_invalid, float_overflow, float_inexact, float_underflow

COUNTS AS

CNT_FLOAT_MUL, CNT_FLOAT_ADD if $u \neq 0$, CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_ADD, FLOAT_SUB_MUL, FLOAT_SUB_MUL_REV, INT_ADD_MUL, §12.4

(FLOAT_APPROX_RESTORE y) $_{64}\ldots{}_{21}00_{16}\,y\,_{12}01_{6}00_{0}$ C

```
$index \leftarrow y_{51\ldots45}$
for $i \in [0\ldots7]$ :
    $\text{rbase}_i \leftarrow y_{2\cdot i}$
    $\text{qbase}_i \leftarrow y_{2\cdot i+1}$
for $i \in [8\ldots15]$ :
    $\text{rbase}_i \leftarrow y_{2\cdot i+2}$
    $\text{qbase}_i \leftarrow y_{2\cdot i+3}$
$\text{rbase}_{16} \leftarrow y_{36}$
$\text{qbase}_{16} \leftarrow y_{37}$
for $i \in [0\ldots2]$ :
    $\text{rslope}_i \leftarrow y_{2\cdot i+38}$
    $\text{qslope}_i \leftarrow y_{2\cdot i+39}$
for $i \in [3\ldots6]$ :
    $\text{rslope}_i \leftarrow y_{2\cdot i+46}$
    $\text{qslope}_i \leftarrow y_{2\cdot i+47}$
$\text{recip base}_{index} \leftarrow \text{rbase}$
$\text{rsqrt base}_{index} \leftarrow \text{qbase}$
$\text{recip slope}_{index} \leftarrow \text{rslope}$
$\text{rsqrt slope}_{index} \leftarrow \text{qslope}$

    {where $y_{16} = parity(\text{rbase}_{0..7}), y_{17} = parity(\text{qbase}_{0..7}),$
$y_{34} = parity(\text{rbase}_{8..15}), y_{35} = parity(\text{qbase}_{8..15}),$
$y_{60} = parity(\text{rbase}_{16}|\text{rslope}), y_{61} = parity(\text{qbase}_{16}|\text{qslope})$}
```

This IPL-privileged operation is used to initialize the floating-point reciprocal and reciprocal square root approximation tables.

RAISES

(nothing)

SEE ALSO

FLOAT_RECIP_APPROX, FLOAT_RSQRT_APPROX

(FLOAT_CMP_TEST t u v) $\quad {}_{64}\ldots{}_{47}t{}_{42}u{}_{37}v{}_{32}11{}_{27}0F{}_{21}\ldots{}_{0}$ A

$t \leftarrow u - v$, floating point

(FLOAT_CMP_TEST x y z) $\quad {}_{64}\ldots{}_{21}x{}_{16}y{}_{11}z{}_{6}0F{}_{0}$ C

$x \leftarrow y - z$, floating point

This operation computes floating-point comparison.

FLOAT_CMP_TEST generates overflow/NaN if either operand is NaN, and raises float_invalid. The condition code is zero if the two operands are equal, negative if the first ($u \vee y$) is less than the second ($v \vee z$), and positive if the first is greater than the second. Carry is set if and only if $v \vee z$ is NaN.

RAISES

float_invalid

COUNTS AS

CNT_FLOAT_ADD, CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_MAX_TEST, FLOAT_MIN_TEST, §5.1

(FLOAT_DIV t u v w) $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,w\,{}_{27}\,\mathrm{1E}\,{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow u \div v * w$, floating point

This operation is used to complete the floating-point division of u by v. The reciprocal in w is a SpecialFloat64 as delivered by FLOAT_ITER. The inexact exception arising from the division of two floating-point numbers will be raised by this operation.

RAISES

float_overflow, float_underflow, float_inexact

COUNTS AS

CNT_FLOAT_DIV, CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_DIV_APPROX, §12.5

(FLOAT_DIV_APPROX *t u v w*) $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,w\,{}_{27}\,\mathbf{3A}\,{}_{21}\cdots{}_{0}$ **A**

exp — unbiased exponent of *u*:
$t - v * w/2^{exp-52}$, floating point, round to nearest

(FLOAT_SQRT_APPROX_TEST *t u v w*) $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,w\,{}_{27}\,\mathbf{3B}\,{}_{21}\cdots{}_{0}$ **A**

exp — unbiased exponent of *u*;
$t - v * w/2^{\lfloor exp/2 \rfloor}$, floating point, round to nearest

These operations perform a floating-point multiply of SpecialFloat64 *w* with Float64 *v*, with round to nearest. They are used for floating-point division and square root computation. A float_invalid exception is raised for 0/0, infinity/infinity, and square root of negative. A float_inexact exception is raised only in conjunction with float_overflow. The condition code produced by FLOAT_SQRT_APPROX_TEST is undefined.

RAISES

float_invalid, float_zero_divide, float_overflow, float_inexact

COUNTS AS

CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_DIV, FLOAT_SQRT, FLOAT_ITER, §12.5

FLOAT_DIV_APPROX

(**FLOAT_DIV_ERROR** $t\ u\ v\ w$) ${}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}\mathrm{3C}_{21}\cdots{}_{0}$ **A**

> *exp* — unbiased exponent of v:
> $t \leftarrow (u - v * w)/2^{exp-52}$, floating point, round to nearest

(**FLOAT_SQRT_ERROR_TEST** $t\ u\ v\ w$) ${}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}\mathrm{3D}_{21}\cdots{}_{0}$ **A**

> *exp* — unbiased exponent of u:
> $t \leftarrow 0.5 * (u - v * w)/2^{\lfloor exp/2 \rfloor}$, floating point, round to nearest

These operations perform a floating-point subtract of u with the product of v and w, using round to nearest. The result is scaled to avoid undesirable overflow or underflow. They are used for floating-point division and square root computation.

FLOAT_DIV_ERROR returns a zero with the sign of u when u is infinity or NaN, or v is infinity, NaN, or zero, or w is infinity or NaN. When the rounded result would be zero and inexact, it is rounded away from zero to produce the minimum denorm floating-point number. FLOAT_SQRT_ERROR_TEST returns a positive zero when v is infinity, NaN, or zero. The condition code produced is undefined.

RAISES

(nothing)

COUNTS AS

CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_DIV, FLOAT_SQRT, FLOAT_ITER, §12.5

(FLOAT_INT *t u***)** $_{64}\cdots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,\mathrm{18}\,{}_{32}\,\mathrm{1E}\,{}_{27}\,\mathrm{08}\,{}_{21}\cdots{}_{0}$ **A**

t — float of integer *u*

This operation converts an integer into a floating-point number, rounding according to the current rounding mode in the ssw.

RAISES

float_inexact

SEE ALSO

FLOAT_UNS

FLOAT_INT_

(FLOAT_ITER $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}38_{21}\cdots{}_{0}$ A

$t \leftarrow u + v * w$, floating point, round to nearest

In use, u and w are an extended precision (SpecialFloat64) reciprocal whose accuracy is increased by cancelling out the relative error given by the floating-point number v. The result is stored as a SpecialFloat64. FLOAT_ITER is used in both floating-point and integer division and square root computations.

RAISES

(nothing)

SEE ALSO

FLOAT_RECIP_APPROX, FLOAT_DIV_APPROX, §12.5

(FLOAT_MAX *t u v***)** $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}13_{27}0E_{21}\cdots{}_{0}$ **A**

$t \leftarrow \max(u, v)$, floating point

(FLOAT_MAX_TEST *t u v***)** $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}13_{27}0F_{21}\cdots{}_{0}$ **A**

$t \leftarrow \max(u, v)$, floating point

These operations select the larger of the two floating-point operands. If both operands are NaN, *u* is selected; if only one is NaN, the other (non-NaN) operand is selected. If both operands are zero, a positive zero is selected if one is present. See §5.1.

FLOAT_MAX_TEST generates overflow/NaN if either operand is NaN. The condition code is zero if the two operands are equal, negative if the first (*u*) is less than the second (*v*), and positive if the first is greater than the second. Carry is set if and only if *v* is NaN.

RAISES

(nothing)

COUNTS AS

CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_MIN, SELECT_FLOAT, INT_MAX

(FLOAT_MIN *t u v*) $_{64}\ldots{}_{47}t_{42}u_{37}v_{32}12_{27}0E_{21}\ldots{}_{0}$ A

$t \leftarrow \min(u, v)$, floating point

(FLOAT_MIN_TEST *t u v*) $_{64}\ldots{}_{47}t_{42}u_{37}v_{32}12_{27}0F_{21}\ldots{}_{0}$ A

$t \leftarrow \min(u, v)$, floating point

These operations select the smaller of the two floating-point operands. If both operands are NaN, *u* is selected; if only one is NaN, the other (non-NaN) operand is selected. See §5.1.

FLOAT_MIN_TEST generates overflow/NaN if either operand is NaN. The condition code is zero if the two operands are equal, negative if the first (*u*) is less than the second (*v*), and positive if the first is greater than the second. Carry is set if and only if *v* is NaN. If both operands are zero, a negative zero is selected if one is present.

RAISES

(nothing)

COUNTS AS

CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_MAX, SELECT_FLOAT, INT_MIN

(**FLOAT_MMAX** *t u v*) $_{64}\ldots_{47}t_{42}u_{37}v_{32}15_{27}0E_{21}\ldots_{0}$ **A**

t — **if** $abs(u) \geq abs(v)$ **then** u **else** v **end**, floating point

(**FLOAT_MMAX_TEST** *t u v*) $_{64}\ldots_{47}t_{42}u_{37}v_{32}15_{27}0F_{21}\ldots_{0}$ **A**

t — **if** $abs(u) \geq abs(v)$ **then** u **else** v **end**, floating point

These operations select the larger in magnitude of the two floating-point operands. If both operands are NaN, *u* is selected; if only one is NaN, the other (non-NaN) operand is selected. If the operands have equal magnitude, *u* is selected.

FLOAT_MMAX_TEST generates overflow/NaN if either operand is NaN. The condition code is zero if the two operands are equal in magnitude, negative if the first (*u*) is smaller than the second (*v*), and positive if the first is larger than the second. Carry is set if and only if *v* is NaN.

RAISES

(nothing)

COUNTS AS

CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_MMIN, SELECT_FLOAT, FLOAT_MAX

FLOAT_MMAX_

(FLOAT_MMIN t u v) ${}_{64}\cdots{}_{47}t{}_{42}u{}_{37}v{}_{32}14{}_{27}\mathrm{0E}{}_{21}\cdots{}_{0}$ A

$t \leftarrow$ if $abs(u) < abs(v)$ **then** u **else** v **end**; floating point

(FLOAT_MMIN_TEST t u v) ${}_{64}\cdots{}_{47}t{}_{42}u{}_{37}v{}_{32}14{}_{27}\mathrm{0F}{}_{21}\cdots{}_{0}$ A

$t \leftarrow$ if $abs(u) < abs(v)$ **then** u **else** v **end**; floating point

These operations select the smaller in magnitude of the two floating-point operands. If both operands are NaN, v is selected: if only one is NaN, the other (non-NaN) operand is selected. If the operands have equal magnitude, v is selected.

FLOAT_MMIN_TEST generates overflow/NaN if either operand is NaN. The condition code is zero if the two operands are equal in magnitude, negative if the first (u) is smaller than the second (v), and positive if the first is larger than the second. Carry is set if and only if v is NaN.

RAISES

(nothing)

COUNTS AS

CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_MMAX, SELECT_FLOAT, FLOAT_MIN

(FLOAT_MUL_LOWER *t u v w*) $_{64}\dots{}_{47}\, t \,_{42}\, u \,_{37}\, v \,_{32}\, w \,_{27}\, 34 \,_{21}\dots{}_{0}$ A

$t — v * w - u$. floating point

This operation performs a floating-point multiply followed by a floating-point subtraction. It differs from FLOAT_SUB_MUL_REV in the response to exceptions. In particular, when u, v, or w is infinity or NaN, the result is zero with no exception. Normally, an infinity or NaN would be returned and an exception possibly raised. In particular, if $v * w$ rounds to infinity and u is that infinity, 0 rather than NaN is generated in t and no exception is raised.

Only one rounding operation is performed, enhancing accuracy and greatly facilitating doubled precision operations.

RAISES

float_underflow, float_overflow, float_inexact

COUNTS AS

CNT_FLOAT_MUL, CNT_FLOAT_ADD if $u \neq 0$, CNT_FLOAT_TOTAL

SEE ALSO

§12.4

(FLOAT_RECIP_APPROX x y) ${}_{64}\cdots{}_{21}x{}_{16}y{}_{11}\,\mathrm{0C}\,{}_{6}\mathrm{00}{}_{0}$ C

exp — unbiased exponent of y;
x — an approximation to $2^{exp-52}/y$, floating point

(FLOAT_RECIP_APPROX_TEST x y) ${}_{64}\cdots{}_{21}x{}_{16}y{}_{11}\,\mathrm{0C}\,{}_{6}\mathrm{01}{}_{0}$ C

exp — unbiased exponent of y;
$x \leftarrow$ an approximation to $2^{exp-52}/y$, floating point

(FLOAT_RSQRT_APPROX x y) ${}_{64}\cdots{}_{21}x{}_{16}y{}_{11}\,\mathrm{0D}\,{}_{6}\mathrm{00}{}_{0}$ C

exp — unbiased exponent of y;
$x \leftarrow$ an approximation to $2^{\lfloor exp/2 \rfloor}/\sqrt{y}$, floating point

(FLOAT_RSQRT_APPROX_TEST x y) ${}_{64}\cdots{}_{21}x{}_{16}y{}_{11}\,\mathrm{0D}\,{}_{6}\mathrm{01}{}_{0}$ C

$exp \leftarrow$ unbiased exponent of y;
$x \leftarrow$ an approximation to $2^{\lfloor exp/2 \rfloor}/\sqrt{y}$, floating point

These operations are used for computing floating-point reciprocals and reciprocal square roots. They perform a table lookup operation followed by a linear interpolation using an adder-multiplier. The table is in an internal format. The approximation is returned as a SpecialFloat64.

In FLOAT_*_APPROX, if the y operand is denormalized, the float_extension exception is raised and y is returned. When the y operand is denormalized with FLOAT_*_APPROX_TEST, they return y, set carry, and raise no exception. In either case, if the y operand is zero, y is returned.

RAISES

float_extension

SEE ALSO

§12.5

FLOAT_APPROX_

(FLOAT_RECIP_ERROR $t\ v\ w$) ${}_{64}\cdots{}_{47}t_{42}\,\mathrm{1E}_{37}\,v_{32}\,w_{27}\,\mathrm{00}_{21}\cdots{}_{0}$ **A**

exp — unbiased exponent of v;
t — $1.0 - v * w/2^{exp-52}$, floating point, round to nearest

(FLOAT_RSQRT_ERROR_TEST $t\ u\ v\ w$) ${}_{64}\cdots{}_{47}t_{42}\,u_{37}\,v_{32}\,w_{27}\,\mathrm{3F}_{21}\cdots{}_{0}$ **A**

exp — unbiased exponent of u:
t — $0.5 * (1.0 - v * w/2^{\lfloor exp/2 \rfloor})$, floating point, round to nearest

These operations are used for computing floating-point reciprocals and square roots. They perform a partial Newton's method iteration using the adder-multiplier, returning the relative error in the SpecialFloat64 reciprocal or reciprocal square root in w compared to the Float64 divisor or square root estimate in v. The condition code produced by FLOAT_RSQRT_ERROR_TEST is undefined.

RAISES

(nothing)

COUNTS AS

CNT_FLOAT_TOTAL

SEE ALSO

INT_RECIP_ERROR, §12.5

(FLOAT_SCALB *t v w*) $_{64}\cdots{}_{47}t_{42}\,18_{37}\,v_{32}\,w_{27}\,00_{21}\cdots{}_{0}$ **A**

$t \leftarrow v * 2^{w}$, floating point

This operation is used to multiply the floating-point number v by a power of two selected by the signed integer in the low 13 bits of w.

RAISES

float_overflow, float_underflow, float_inexact

COUNTS AS

CNT_FLOAT_TOTAL

SEE ALSO

INT_LOGB

(FLOAT_SQRT t u v w) ${}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}w_{27}\mathrm{1F}_{21}\ldots{}_{0}$ **A**

$t \leftarrow u + v * w$. floating point

This operation is used to complete floating-point square root of the floating-point number in u. using the SpecialFloat64 reciprocal in w. The inexact exception arising from computing the square root of a floating-point number is raised by this operation.

RAISES

float_inexact

COUNTS AS

CNT_FLOAT_SQRT, CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_RSQRT_APPROX, §12.5

(**FLOAT_SUB** *t u v*) $\quad {}_{64}\cdots{}_{47}t{}_{42}u{}_{37}v{}_{32}\mathrm{11}{}_{27}\mathrm{0E}{}_{21}\cdots{}_{0}$ A

$t \leftarrow u - v$, floating point

(**FLOAT_SUB** *x y z*) $\quad {}_{64}\cdots{}_{21}x{}_{16}y{}_{11}z{}_{6}\mathrm{0E}{}_{0}$ C

$x \leftarrow y - z$, floating point

This operation computes floating-point subtraction.

RAISES

float_invalid, float_overflow, float_inexact

COUNTS AS

CNT_FLOAT_ADD, CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_SUB_MUL, FLOAT_SUB_MUL_REV

FLOAT_SUB_

(**FLOAT_SUB_MUL** $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}t{}_{42}u{}_{37}v{}_{32}w{}_{27}32{}_{21}\cdots{}_{0}$ **A**

t — $u - v * w$, floating point

(**FLOAT_SUB_MUL_REV** $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}t{}_{42}u{}_{37}v{}_{32}w{}_{27}36{}_{21}\cdots{}_{0}$ **A**

t — $v * w - u$, floating point

These operations perform a floating-point multiply followed by a floating-point subtraction. If u is register 0. only a floating-point multiply is performed (to preserve the sign of a zero result).

RAISES

float_invalid, float_overflow, float_inexact, float_underflow

COUNTS AS

CNT_FLOAT_MUL, CNT_FLOAT_ADD if $u \neq 0$, CNT_FLOAT_TOTAL

SEE ALSO

FLOAT_ADD_MUL, FLOAT_SUB

(FLOAT_UNS *t u*) ${}_{64}\cdots{}_{47}t_{42}u_{37}18_{32}1F_{27}08_{21}\cdots{}_{0}$ **A**

t — float of unsigned integer *u*

This operation converts unsigned integers into floating-point numbers, rounding according to the current rounding mode in the SSW.

RAISES

float_inexact

SEE ALSO

FLOAT_INT

FLOAT_UNS_

(INT_CEIL *t u*) $_{64}\cdots_{47}t_{42}u_{37}18_{32}07_{27}08_{21}\cdots_{0}$ **A**

t — ceiling of float *u*

(INT_CEIL_TEST *t u*) $_{64}\cdots_{47}t_{42}u_{37}18_{32}07_{27}09_{21}\cdots_{0}$ **A**

t ← ceiling of float *u*

(INT_CHOP *t u*) $_{64}\cdots_{47}t_{42}u_{37}18_{32}05_{27}08_{21}\cdots_{0}$ **A**

t — integer chop of float *u*

(INT_CHOP_TEST *t u*) $_{64}\cdots_{47}t_{42}u_{37}18_{32}05_{27}09_{21}\cdots_{0}$ **A**

t ← integer chop of float *u*

(INT_FLOOR *t u*) $_{64}\cdots_{47}t_{42}u_{37}18_{32}06_{27}08_{21}\cdots_{0}$ **A**

t ← floor of float *u*

(INT_FLOOR_TEST *t u*) $_{64}\cdots_{47}t_{42}u_{37}18_{32}06_{27}09_{21}\cdots_{0}$ **A**

t — floor of float *u*

(INT_NEAR *t u*) $_{64}\cdots_{47}t_{42}u_{37}18_{32}04_{27}08_{21}\cdots_{0}$ **A**

t ← integer nearest float *u*

(INT_NEAR_TEST *t u*) $_{64}\cdots_{47}t_{42}u_{37}18_{32}04_{27}09_{21}\cdots_{0}$ **A**

t ← integer nearest float *u*

(INT_ROUND *t u*) $_{64}\cdots_{47}t_{42}u_{37}18_{32}0D_{27}08_{21}\cdots_{0}$ **A**

t ← integer round of float *u*

(INT_ROUND_TEST *t u*) $_{64}\cdots_{47}t_{42}u_{37}18_{32}0D_{27}09_{21}\cdots_{0}$ **A**

t ← integer round of float *u*

These operations convert floats into signed integers. The roundings are directed as in IEEE Standard 754. INT_ROUND uses the rounding mode in the ssw.

A float_invalid exception is raised when the result is not a representable signed integer. In these cases the result is reduced modulo 2^{64}.

The _TEST versions of these operations never generate carry or overflow/NaN.

RAISES

float_invalid, float_inexact

SEE ALSO

FLOAT_INT, FLOAT_ UNS_

(INT_ADD t u v) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}\text{1C}_{27}\text{0E}_{21}\cdots{}_{0}$ **A**

$t \leftarrow u + v$, integer

(INT_ADD x y z) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}z_{6}\text{24}_{0}$ **C**

$x \leftarrow y + z$, integer

(INT_ADD_TEST t u v) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}\text{1C}_{27}\text{0F}_{21}\cdots{}_{0}$ **A**

$t \leftarrow u + v$, integer

(INT_ADD_TEST x y z) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}z_{6}\text{25}_{0}$ **C**

$x \leftarrow y + z$, integer

These operations perform two's-complement and unsigned integer addition.

The resulting condition code from the _TEST version of this operation has its two's-complement definition.

RAISES

(nothing)

SEE ALSO

INT_ADD_IMM

INT_ADD_

(INT_ADD_IMM *t u value*) ${}_{64}\cdots{}_{47}t_{42}u_{37}0_{36}\,bvalue_{27}20_{21}\cdots{}_{0}$ **A**

$t \leftarrow u + {}'value$, integer
{**where** $'value \in [1 \ldots 512]$, $'bvalue = {}'value - 1$}

(INT_ADD_IMM *x y value*) ${}_{64}\cdots{}_{21}x_{16}y_{11}\,bvalue_{6}04_{0}$ **C**

$x \leftarrow y + {}'value$, integer
{**where** $'value \in [1 \ldots 32]$, $'bvalue = {}'value - 1$}

(INT_ADD_IMM_TEST *t u value*) ${}_{64}\cdots{}_{47}t_{42}u_{37}0_{36}\,bvalue_{27}21_{21}\cdots{}_{0}$ **A**

$t \leftarrow u + {}'value$, integer
{**where** $'value \in [1 \ldots 512]$, $'bvalue = {}'value - 1$}

(INT_ADD_IMM_TEST *x y value*) ${}_{64}\cdots{}_{21}x_{16}y_{11}\,bvalue_{6}05_{0}$ **C**

$x \leftarrow y + {}'value$, integer
{**where** $'value \in [1 \ldots 32]$, $'bvalue = {}'value - 1$}

These operations effectively add a constant between 1 and 32(512) to $y(u)$, storing it in $x(t)$.

The resulting condition code from the _TEST version of this operation has its two's-complement definition.

RAISES

(nothing)

SEE ALSO

INT_ADD, INT_SUB_IMM

(INT_ADD_MUL *t u v w*) $\quad {}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}w_{27}28_{21}\ldots{}_{0}$ **A**

$t \leftarrow u + v * w$, integer

(INT_ADD_MUL_TEST *t u v w*) $\quad {}_{64}\ldots{}_{47}t_{42}u_{37}v_{32}w_{27}29_{21}\ldots{}_{0}$ **A**

$t \leftarrow u + v * w$, integer

These operations perform two's-complement multiplication and addition. A multiply is accomplished by letting u be register 0.

The _TEST versions of these operations never generate carry or overflow/NaN, despite the fact that the multiply or the add might overflow.

If v or w is outside $[-2^{53} \ldots 2^{53} - 1]$, the float_extension exception is raised and the result in t may be incorrect.

RAISES

float_extension

SEE ALSO

UNS_ADD_MUL_UPPER, INT_ADD, INT_SUB_MUL, INT_SUB_MUL_REV

INT_ADD_MUL_

(INT_DIV_CHOP *t u v w*) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}\,18\,_{21}\cdots{}_{0}$ A

exp — unbiased exponent of w
$temp$ — $v * w/2^{exp}$, round to zero
t — $temp * 2^{exp}$, round to zero

(INT_DIV_CHOP_TEST *t u v w*) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}\,19\,_{21}\cdots{}_{0}$ A

exp — unbiased exponent of w
$temp$ — $v * w/2^{exp}$, round to zero
t — $temp * 2^{exp}$, round to zero

(INT_DIV_FLOOR *t u v w*) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}\,1A\,_{21}\cdots{}_{0}$ A

exp — unbiased exponent of w
$temp$ — $v * w/2^{exp}$, round to zero
t — $temp * 2^{exp}$, round to floor

(INT_DIV_FLOOR_TEST *t u v w*) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}\,1B\,_{21}\cdots{}_{0}$ A

exp — unbiased exponent of w
$temp$ — $v * w/2^{exp}$, round to zero
t — $temp * 2^{exp}$, round to floor

These operations are the last step in integer division. The product of the integer v and SpecialFloat64 w is shifted right according to the exponent of w and rounded, producing an integer.

The _TEST versions of these operations generate carry when the quotient is not exact, i.e. when the division by 2^{-exp} yields a non-zero remainder.

If v is outside $[-2^{53} \ldots 2^{53} - 1]$, the float_extension exception is raised and the result in t may be incorrect.

Although register u is not used in the current hardware implementation, the software requires u to contain the denominator in order to properly handle float_extension exceptions.

RAISES

float_extension

SEE ALSO

UNS_DIV, §12.6

(**INT_FETCH_ADD_AC_DISP** *r s ac disp*) $_{64}\cdots{}_{61}r_{56}s_{51}C_{47}\cdots{}_{21}ac_{16}sdisp_{5}11_{0}$ MC

$temp \leftarrow$ (word at $s +$ *'disp mod* 2^{48});
(word at $s +$ *'disp mod* 2^{48}) $\leftarrow r +$ (word at $s +$ *'disp mod* 2^{48}), with *'ac*;
$r \leftarrow temp$
{where *'disp* $\in [0 \ldots 16383]$, *'sdisp* = *'disp*/8}

(**INT_FETCH_ADD_AC_INDEX** *r s ac y*) $_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}ac_{16}y_{11}10_{6}39_{0}$ MC

$temp \leftarrow$ (word at $s + 8 * y$ *mod* 2^{48});
(word at $s + 8 * y$ *mod* 2^{48}) $\leftarrow r +$ (word at $s + 8 * y$ *mod* 2^{48}), with *'ac*;
$r \leftarrow temp$

(**INT_FETCH_ADD_DISP** *r s disp*) $_{64}\cdots{}_{61}r_{56}s_{51}C_{47}\cdots{}_{21}sdisp_{5}10_{0}$ MC

$temp \leftarrow$ (word at $s +$ *'disp mod* 2^{48});
(word at $s +$ *'disp mod* 2^{48}) $\leftarrow r +$ (word at $s +$ *'disp mod* 2^{48});
$r \leftarrow temp$
{where *'disp* $\in [0 \ldots 524287]$, *'sdisp* = *'disp*/8}

(**INT_FETCH_ADD_INDEX** *r s y*) $_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}00_{16}y_{11}10_{6}38_{0}$ MC

$temp \leftarrow$ (word at $s + 8 * y$ *mod* 2^{48});
(word at $s + 8 * y$ *mod* 2^{48}) $\leftarrow r +$ (word at $s + 8 * y$ *mod* 2^{48});
$r \leftarrow temp$;

These operations generally behave like a LOAD operation followed by a STORE operation with respect to access control. If *ac* is present, it is used; otherwise the access control field of *s* is used.

RAISES

data_hw_error. data_prot, data_alignment, data_blocked

COUNTS AS

CNT_INT_FETCH_ADD

SEE ALSO

LOAD, INT_ADD

(INT_IMM *t value*) $_{64}\ldots_{47}t_{42}\,\mathit{value}_{27}\,02_{21}\ldots_{0}$ **A**

$t \leftarrow$ value
{where $\mathit{value} \in [-2^{14} \ldots 2^{14} - 1]$}

This operation loads a signed immediate constant into register t.

RAISES

(nothing)

SEE ALSO

BIT_MASK

(INT_LOADB *r s*) ${}_{64}\cdots{}_{61}r_{56}s_{51}3_{47}\cdots{}_{0}$ M

$r \leftarrow$ sign extend(byte at s), with FE_NORMAL

(INT_LOADB_AC_DISP *r s ac disp*) ${}_{64}\cdots{}_{61}r_{56}s_{51}C_{47}\cdots{}_{21}ac_{16}disp_{2}3_{0}$ MC

$r \leftarrow$ sign extend(byte at $s +$ *'disp mod* 2^{48}), with *'ac*
{**where** *'disp* $\in [0 \ldots 16383]$}

(INT_LOADB_AC_INDEX *r s ac y*) ${}_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}ac_{16}y_{11}16_{6}39_{0}$ MC

$r \leftarrow$ sign extend(byte at $s + y$ *mod* 2^{48}), with *'ac*

(INT_LOADB_DISP *r s disp*) ${}_{64}\cdots{}_{61}r_{56}s_{51}C_{47}\cdots{}_{21}disp_{2}2_{0}$ MC

$r \leftarrow$ sign extend(byte at $s +$ *'disp mod* 2^{48}), with FE_NORMAL
{**where** *'disp* $\in [0 \ldots 524287]$}

(INT_LOADB_INDEX *r s y*) ${}_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}00_{16}y_{11}16_{6}38_{0}$ MC

$r \leftarrow$ sign extend(byte at $s + y$ *mod* 2^{48}), with FE_NORMAL

These operations load a signed byte from memory. The fe_control is taken from the *ac* field if present, or forced to FE_NORMAL. If *ac* is present, its forward, data trap0, and data trap1 disable bits are used; otherwise those of *s* are used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_LOAD

SEE ALSO

STOREB, UNS_LOADB

INT_LOADB_

(INT_LOADH *r s*) $_{64}\cdots{}_{61}r_{56}s_{51}1_{47}\cdots{}_{0}$ M

$r \leftarrow$ sign extend(halfword at s), with FE_NORMAL

(INT_LOADH_AC_DISP *r s ac disp*) $_{64}\cdots{}_{61}r_{56}s_{51}C_{47}\cdots{}_{21}ac_{16}sdisp_{4}9_{0}$ MC

$r \leftarrow$ sign extend(halfword at $s +$ $'disp$ *mod* 2^{48}), with $'ac$
{**where** $'disp \in [0 \ldots 16383]$, $'sdisp = 'disp/4$}

(INT_LOADH_AC_INDEX *r s ac y*) $_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}ac_{16}y_{11}12_{6}39_{0}$ MC

$r \leftarrow$ sign extend(halfword at $s + 4 * y$ *mod* 2^{48}), with $'ac$

(INT_LOADH_DISP *r s disp*) $_{64}\cdots{}_{61}r_{56}s_{51}C_{47}\cdots{}_{21}sdisp_{4}8_{0}$ MC

$r \leftarrow$ sign extend(halfword at $s +$ $'disp$ *mod* 2^{48}), with FE_NORMAL
{**where** $'disp \in [0 \ldots 524287]$, $'sdisp = 'disp/4$}

(INT_LOADH_INDEX *r s y*) $_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}00_{16}y_{11}12_{6}38_{0}$ MC

$r \leftarrow$ sign extend(halfword at $s + 4 * y$ *mod* 2^{48}), with FE_NORMAL

These operations load a signed halfword from memory. The fe_control is taken from the *ac* field if present, or forced to FE_NORMAL. If *ac* is present, its forward, data trap0, and data trap1 disable bits are used; otherwise those of *s* are used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_LOAD

SEE ALSO

STOREH, UNS_LOADH

(INT_LOADQ *r s*) $\;{}_{64}\cdots{}_{61}r{}_{56}s{}_{51}2{}_{47}\cdots{}_{0}$ M

r — sign extend(quarterword at *s*), with FE_NORMAL

(INT_LOADQ_AC_DISP *r s ac disp*) $\;{}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{C}{}_{47}\cdots{}_{21}ac{}_{16}sdisp{}_{3}5{}_{0}$ MC

r — sign extend(quarterword at $s +$ *'disp mod* 2^{48}), with *'ac*
{**where** *'disp* $\in [0 \ldots 16383]$, *'sdisp* = *'disp*/2}

(INT_LOADQ_AC_INDEX *r s ac y*) $\;{}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}ac{}_{16}y{}_{11}14{}_{6}39{}_{0}$ MC

r — sign extend(quarterword at $s + 2 * y$ *mod* 2^{48}), with *'ac*

(INT_LOADQ_DISP *r s disp*) $\;{}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{C}{}_{47}\cdots{}_{21}sdisp{}_{3}4{}_{0}$ MC

r — sign extend(quarterword at $s +$ *'disp mod* 2^{48}), with FE_NORMAL
{**where** *'disp* $\in [0 \ldots 524287]$, *'sdisp* = *'disp*/2}

(INT_LOADQ_INDEX *r s y*) $\;{}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}0{}_{16}y{}_{11}14{}_{6}38{}_{0}$ MC

r — sign extend(quarterword at $s + 2 * y$ *mod* 2^{48}), with FE_NORMAL

These operations load a signed quarterword from memory. The fe_control is taken from the *ac* field if present, or forced to FE_NORMAL. If *ac* is present, its forward, data trap0, and data trap1 disable bits are used; otherwise those of *s* are used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_LOAD

SEE ALSO

STOREQ, UNS_LOADQ

(INT_LOGB x y) $_{64}\ldots{}_{21}x_{16}y_{11}\mathrm{0B}_{6}\mathrm{00}_{0}$ C

x — unbiased exponent of float y

(INT_LOGB_TEST x y) $_{64}\ldots{}_{21}x_{16}y_{11}\mathrm{0B}_{6}\mathrm{01}_{0}$ C

x — unbiased exponent of float y

These operations determine the floor of the base 2 logarithm of a floating-point number.

For denormalized y, x will take on values less than the minimum exponent so that scalb(x, -logb(x)) is always less than two and greater than or equal to one. When y is infinity or NaN, the maximum positive integer is returned. When y is zero. the minimum negative integer is returned.

INT_LOGB_TEST never generates overflow/NaN, and generates carry if y is infinity, NaN or zero.

RAISES

(nothing)

(INT_MAX t u v) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}\mathrm{1F}_{27}\mathrm{0E}_{21}\cdots{}_{0}$ **A**

$t \leftarrow \max(u, v)$; integer

(INT_MAX_TEST t u v) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}\mathrm{1F}_{27}\mathrm{0F}_{21}\cdots{}_{0}$ **A**

$t \leftarrow \max(u, v)$; integer

These operations select the larger of the two integer operands.

INT_MAX_TEST never generates overflow/NaN, and generates carry if u is selected, meaning $u \geq v$.

RAISES

(nothing)

SEE ALSO

INT_MIN, SELECT_INT, FLOAT_MAX

(INT_MEM_ADD_AC_DISP *r s ac disp*) $\;{}_{64}\cdots{}_{51}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}ac{}_{16}sdisp{}_{5}13{}_{0}$ MC

(word at $s +$ '$disp$ mod 2^{48}) $-$ r $+$ (word at $s +$ '$disp$ mod 2^{48}), with 'ac;

{**where** '$disp \in [0\ldots16383]$, '$sdisp$ = '$disp/8$}

(INT_MEM_ADD_AC_INDEX *r s ac y*) $\;{}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}ac{}_{16}y{}_{11}13{}_{6}39{}_{0}$ MC

(word at $s + 8 * y$ mod 2^{48}) $-$ r $+$ (word at $s + 8 * y$ mod 2^{48}), with 'ac;

(INT_MEM_ADD_DISP *r s disp*) $\;{}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}sdisp{}_{5}12{}_{0}$ MC

(word at $s +$ '$disp$ mod 2^{48}) $-$ r $+$ (word at $s +$ '$disp$ mod 2^{48});

{**where** '$disp \in [0\ldots524287]$, '$sdisp$ = '$disp/8$}

(INT_MEM_ADD_INDEX *r s y*) $\;{}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}00{}_{16}y{}_{11}13{}_{6}38{}_{0}$ MC

(word at $s + 8 * y$ mod 2^{48}) $-$ r $+$ (word at $s + 8 * y$ mod 2^{48});

These operations generally behave like a LOAD operation followed by a STORE operation with respect to access control. Unlike INT_FETCH_ADD, the *r* register is not modified. If *ac* is present, it is used; otherwise the access control field of *s* is used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

SEE ALSO

LOAD, INT_ADD

(INT_MIN t u v) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}\mathrm{1E}_{27}\mathrm{0E}_{21}\cdots{}_{0}$ A

$t \leftarrow \min(u, v)$, integer

(INT_MIN_TEST t u v) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}\mathrm{1E}_{27}\mathrm{0F}_{21}\cdots{}_{0}$ A

$t \leftarrow \min(u, v)$, integer

These operations select the smaller of the two integer operands. INT_MIN_TEST never generates overflow/NaN, and generates carry if v is selected, meaning $u \geq v$.

RAISES

(nothing)

SEE ALSO

INT_MAX. SELECT_INT, FLOAT_MIN

INT_MIN_

(INT_RECIP_APPROX x y) $\quad {}_{64}\cdots{}_{21}\,x\,{}_{16}\,y\,{}_{11}\,08\,{}_{6}\,00\,{}_{0}$ C

x — an approximation to $1/y$. floating point

This operation is used to compute an integer reciprocal. It performs a table lookup operation, followed by a linear interpolation using an adder-multiplier. The result is returned as a SpecialFloat64. If the y operand is zero. the float_extension exception is raised.

RAISES

float_extension

SEE ALSO

FLOAT_RECIP_APPROX, §12.6

(INT_RECIP_ERROR *t v w*)

t — $1.0 - v * w$, floating point, round to nearest $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,\mathrm{1C}\,{}_{37}\,v\,{}_{32}\,w\,{}_{27}\,\mathrm{00}\,{}_{21}\cdots{}_{0}$ **A**

This operation is used to perform integer division. The _ERROR_ operations perform a partial Newton's method iteration using the adder-multiplier. Note that v is a Float64, while w is used as a SpecialFloat64 and t is returned as a Float64.

RAISES

(nothing)

SEE ALSO

INT_DIV_CHOP, INT_DIV_FLOOR, INT_RECIP_APPROX, §12.6

(INT_RECIP_SHIFT x y) $_{64}\cdots{}_{21}x_{16}y_{11}\,\text{0E}_{6}\,\text{00}_{0}$ C

$x \leftarrow log_2 abs(y)$, round to ceiling

(INT_RECIP_SHIFT_TEST x y) $_{64}\cdots{}_{21}x_{16}y_{11}\,\text{0E}_{6}\,\text{01}_{0}$ C

$x \leftarrow log_2 abs(y)$, round to ceiling

These operations are used to compute integer reciprocals. They compute the ceiling of the base 2 logarithm of the absolute value of y. When y is zero, x is set to -1.

RAISES

(nothing)

SEE ALSO

INT_DIV_CHOP, §12.6

(INT_RSQRT_APPROX x y) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}09_{6}00_{0}$ C

x — an approximation to $1/\sqrt{y}$. floating point

This operation is used to compute the reciprocal square root of a denormalized number. It performs a table lookup operation. followed by a linear interpolation using an adder-multiplier. The result is returned as a SpecialFloat64. If the absolute value of y is outside the range 2^{64} to 1, it is effectively scaled into that range.

RAISES

(nothing)

SEE ALSO

FLOAT_RSQRT_APPROX. §12.5

(INT_SHIFT_RIGHT t v w) ${}_{64}\cdots{}_{47}t{}_{42}\mathrm{1A}{}_{37}v{}_{32}w{}_{27}\mathrm{00}{}_{21}\cdots{}_{0}$ A

$t \leftarrow v \gg_a w$

(INT_SHIFT_RIGHT x y z) ${}_{64}\cdots{}_{21}x{}_{16}y{}_{11}z{}_{6}\mathrm{1E}{}_{0}$ C

$x \leftarrow y \gg_a z$

(INT_SHIFT_RIGHT_TEST t v w) ${}_{64}\cdots{}_{47}t{}_{42}\mathrm{1A}{}_{37}v{}_{32}w{}_{27}\mathrm{01}{}_{21}\cdots{}_{0}$ A

$t \leftarrow v \gg_a w$

(INT_SHIFT_RIGHT_TEST x y z) ${}_{64}\cdots{}_{21}x{}_{16}y{}_{11}z{}_{6}\mathrm{1F}{}_{0}$ C

$x \leftarrow y \gg_a z$

These operations do an arithmetic shift right, filling bits on the left with copies of the sign bit. Unsigned shift counts in w/z are taken modulo 64.

The _TEST version generates carry if a 1-bit is shifted out of v or y and never generates overflow/NaN.

RAISES

(nothing)

SEE ALSO

UNS_SHIFT_RIGHT, SHIFT_PAIR_RIGHT, SHIFT_LEFT, INT_DIV_FLOOR

(INT_SUB $t\ u\ v$) $\quad {}_{64}\cdots{}_{47}t{}_{42}u{}_{37}v{}_{32}\mathrm{1D}{}_{27}\mathrm{0E}{}_{21}\cdots{}_{0}$ A

$t \leftarrow u - v$, integer

(INT_SUB $x\ y\ z$) $\quad {}_{64}\cdots{}_{21}x{}_{16}y{}_{11}z{}_{6}26{}_{0}$ C

$x \leftarrow y - z$, integer

(INT_SUB_TEST $t\ u\ v$) $\quad {}_{64}\cdots{}_{47}t{}_{42}u{}_{37}v{}_{32}\mathrm{1D}{}_{27}\mathrm{0F}{}_{21}\cdots{}_{0}$ A

$t \leftarrow u - v$, integer

(INT_SUB_TEST $x\ y\ z$) $\quad {}_{64}\cdots{}_{21}x{}_{16}y{}_{11}z{}_{6}27{}_{0}$ C

$x \leftarrow y - z$, integer

These operations do two's-complement integer and unsigned subtraction.

The resulting condition code from the _TEST version of this operation has its two's-complement definition.

RAISES

(nothing)

SEE ALSO

INT_SUB_IMM, UNS_SUB_CARRY_TEST

(INT_SUB_IMM *t u value*) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}1_{36}bvalue_{27}20_{21}\cdots{}_{0}$ **A**

$t \leftarrow u - {}'value$, integer
{**where** $'value \in [1 \ldots 512]$, $'bvalue = {}'value - 1$}

(INT_SUB_IMM *x y value*) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}bvalue_{6}06_{0}$ **C**

$x \leftarrow y - {}'value$, integer
{**where** $'value \in [1 \ldots 32]$, $'bvalue = {}'value - 1$}

(INT_SUB_IMM_TEST *t u value*) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}1_{36}bvalue_{27}21_{21}\cdots{}_{0}$ **A**

$t \leftarrow u - {}'value$, integer
{**where** $'value \in [1 \ldots 512]$, $'bvalue = {}'value - 1$}

(INT_SUB_IMM_TEST *x y value*) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}bvalue_{6}07_{0}$ **C**

$x \leftarrow y - {}'value$, integer
{**where** $'value \in [1 \ldots 32]$, $'bvalue = {}'value - 1$}

These operations effectively subtract a constant between 1 and 32(512) to $y(u)$, storing it in $x(t)$.

The resulting condition code from the _TEST version of this operation has its two's-complement definition.

RAISES

(nothing)

SEE ALSO

INT_SUB_IMM, INT_ADD_IMM

(INT_SUB_MUL $t\ u\ v\ w$) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}\text{2A}_{21}\cdots{}_{0}$ **A**

$t - u - v * w$, integer

(INT_SUB_MUL_TEST $t\ u\ v\ w$) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}\text{2B}_{21}\cdots{}_{0}$ **A**

$t - u - v * w$, integer

(INT_SUB_MUL_REV $t\ u\ v\ w$) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}\text{2E}_{21}\cdots{}_{0}$ **A**

$t - -u + v * w$, integer

(INT_SUB_MUL_REV_TEST $t\ u\ v\ w$) $_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}\text{2F}_{21}\cdots{}_{0}$ **A**

$t - -u + v * w$, integer

These operations do two's-complement multiplication followed by subtraction.

The _TEST versions of these operations never generate carry or overflow/NaN, despite the fact that the multiply or the add might overflow.

If v or w is outside $[-2^{53} \ldots 2^{53} - 1]$, the float_extension exception is raised.

RAISES

float_extension

SEE ALSO

INT_SUB, INT_ADD_MUL

(JUMP *mask cn tn*)

```
...  mask   cn  F 1 6 tn      C
64  21     13  11 7 6 3  0
```

```
if CV_cn ∈ 'mask then
    SSW.pc ← TN;
end
```

(JUMP_OFTEN *mask cn tn*)

```
...  mask   cn  F 0 7 tn      C
64  21     13  11 7 5 3  0
```

```
if CV_cn ∈ 'mask then
    SSW.pc ← TN;
else
    stop lookahead;
end
```

(JUMP_SELDOM *mask cn tn*)

```
...  mask   cn  F 1 7 tn      C
64  21     13  11 7 6 3  0
```

```
if CV_cn ∈ 'mask then
    SSW.pc ← TN;
    stop lookahead;
end
```

These operations do conditional branches if the selected condition code is a member of the condition *mask*. If taken, the branch goes to the location previously loaded into target register TN by a TARGET operation. The value *mask* is an eight-bit condition mask of type CondMask, described in §4. If *mask* is empty, then the test is false.

The trap mask SSW.tm, mode bits SSW.md, and condition vector CV are unaffected.

A JUMP_OFTEN operation is intended for branches that will be taken frequently, such as the backwards branch in a loop or a branch over an error condition. Lookahead is stopped if the JUMP_OFTEN fails. Lookahead is stopped if the JUMP_SELDOM succeeds.

JUMP_OFTEN and JUMP_SELDOM can be monitored via CNT_JUMP_EXPECTED and CNT_JUMP_UNEXPECTED to observe branch prediction accuracy. JUMP only counts toward CNT_TRANSFER_TOTAL.

RAISES

(nothing)

COUNTS AS

CNT_JUMP_EXPECTED if expected path taken, CNT_JUMP_UNEXPECTED if unexpected path taken, CNT_TRANSFER_TOTAL

SEE ALSO

SKIP

(LEVEL_ENTER *lev*) $_{64}\cdots{}_{21}00_{16}01_{13}lev_{11}0_{7}0_{6}6_{3}0_{0}$ C

```
if (LEVEL = 'lev) then
    LEVEL ← program map execute protection level;
    SSW.ssw_override ← true;
    suppress program protection exception
else
    raise program protection exception
end
```

(LEVEL_RTN *lev tn*) $_{64}\cdots{}_{21}00_{16}01_{13}lev_{11}F_{7}0_{6}6_{3}tn_{0}$ C

```
if LEVEL ≥ 'lev then
    SSW.pc ← tn;
    SSW.ssw_override ← false;
    LEVEL ← 'lev;
else
    raise privileged operation exception
end
```

These operations change the privilege level of a stream (see §8.1). The LEVEL_ENTER operation is normally placed at privileged entry points, with a matching LEVEL_RTN at the exit. If a LEVEL_ENTER is executed from the wrong privilege level a program protection exception will be raised, whether the privilege level of the stream matched the program map execute protection level or not. If a LEVEL_RTN attempts to raise the privilege level, a privileged operation exception will be raised.

Lookahead is disabled when LEVEL_ENTER sets ssw_override. However, lookahead beyond a LEVEL_ENTER still may result in lost exception detail. Lookahead is not disabled for LEVEL_-RTN.

RAISES

privileged, prog_prot

COUNTS AS

CNT_LEVEL if LEVEL_ENTER

SEE ALSO

DOMAIN_LEAVE

LEVEL_

(**LOAD** *r s*) ${}_{64}\cdots{}_{61}r_{56}s_{51}4_{47}\cdots{}_{0}$ M

$r \leftarrow$ (word at s), with FE_NORMAL

(**LOAD_AC_DISP** *r s ac disp*) ${}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{D}_{47}\cdots{}_{21}ac_{16}sdisp_{5}11_{0}$ MC

$r \leftarrow$ (word at $s +$ '$disp$ mod 2^{48}), with 'ac
{**where** '$disp \in [0 \ldots 16383]$, '$sdisp =$ '$disp/8$}

(**LOAD_AC_INDEX** *r s ac y*) ${}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{F}_{47}\cdots{}_{21}ac_{16}y_{11}08_{6}39_{0}$ MC

$r \leftarrow$ (word at $s + 8 * y$ mod 2^{48}), with 'ac

(**LOAD_DISP** *r s disp*) ${}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{D}_{47}\cdots{}_{21}sdisp_{5}10_{0}$ MC

$r \leftarrow$ (word at $s +$ '$disp$ mod 2^{48}), with FE_NORMAL
{**where** '$disp \in [0 \ldots 524287]$, '$sdisp =$ '$disp/8$}

(**LOAD_INDEX** *r s y*) ${}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{F}_{47}\cdots{}_{21}00_{16}y_{11}08_{6}38_{0}$ MC

$r \leftarrow$ (word at $s + 8 * y$ mod 2^{48}), with FE_NORMAL

These operations load a word from memory. The fe_control is taken from the *ac* field if present, or forced to FE_NORMAL. If *ac* is present, its forward, data trap0, and data trap1 disable bits are used; otherwise those of *s* are used. They are used to load floating-point numbers and 64-bit signed and unsigned integers.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_LOAD

SEE ALSO

STORE

(**LOAD_FE** *r s*)

$_{64}\cdots{}_{61}\,r\,{}_{56}\,s\,{}_{51}\,0\,{}_{47}\cdots{}_{0}$ M

r ← (word at *s*)

This operation loads a word from memory, obeying the fe_control in the pointer. It is used to load floating-point numbers, and 64-bit signed and unsigned integers. This operation allows synchronizing loads to be performed without using explicit access control in the operation.

RAISES

data_hw_error. data_prot, data_alignment, data_blocked

COUNTS AS

CNT_LOAD

SEE ALSO

STORE.PTR_SET_AC

LOAD_FE_

(LOGICAL_ALLONE *t mask cn*) $_{64}\dots_{47}t_{42}\,\mathrm{0B}_{37}\,mask_{29}\,cn_{27}\,\mathrm{00}_{21}\dots_{0}$ **A**

$t \leftarrow$ if $\mathrm{cv}_{cn} \in$ '*mask* then -1 else 0 end

(LOGICAL_ALLONE_TEST *t mask cn*) $_{64}\dots_{47}t_{42}\,\mathrm{0B}_{37}\,mask_{29}\,cn_{27}\,\mathrm{01}_{21}\dots_{0}$ **A**

$t \leftarrow$ if $\mathrm{cv}_{cn} \in$ '*mask* then -1 else 0 end

These operations convert a condition code into a logical value in all the bits in *t*. In contrast, LOGICAL_ONE produces a single-bit logical value.

The fields *mask* and *cn* together refer to cv_{cn}, a condition code that was previously generated. The value *mask* is an eight-bit wide condition mask of type CondMask, described in §4.

The resulting condition code from the _TEST version of this operation has its two's-complement definition.

RAISES

(nothing)

SEE ALSO

LOGICAL_ONE, SELECT_INT

(LOGICAL_ONE *t mask cn*) $_{64}\cdots{}_{47}\,t\,{}_{42}\,\mathrm{0A}\,{}_{37}\,mask\,{}_{29}\,cn\,{}_{27}\,\mathrm{00}\,{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ **if** $\mathrm{CV}_{cn} \in$ '*mask* **then** 1 **else** 0 **end**

(LOGICAL_ONE_TEST *t mask cn*) $_{64}\cdots{}_{47}\,t\,{}_{42}\,\mathrm{0A}\,{}_{37}\,mask\,{}_{29}\,cn\,{}_{27}\,\mathrm{01}\,{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ **if** $\mathrm{CV}_{cn} \in$ '*mask* **then** 1 **else** 0 **end**

These operations convert a condition code into a single-bit logical value in bit 0 of *t*. In contrast, LOGICAL_ALLONE produces a full-word logical value.

The fields *mask* and *cn* together refer to CV_{cn}, a condition code that was previously generated. The value *mask* is an eight-bit wide condition mask of type CondMask, described in §4.

The resulting condition code from the _TEST version of this operation has its two's-complement definition.

RAISES

(nothing)

SEE ALSO

LOGICAL_ALLONE, SELECT_INT

(NOP) $_{64}\cdots{}_{61}00_{56}00_{51}6_{47}\cdots{}_{0}$ M

no action taken

(NOP) $_{64}\cdots{}_{47}00_{42}00_{37}00_{32}00_{27}02_{21}\cdots{}_{0}$ A

no action taken

(NOP) $_{64}\cdots{}_{21}00_{16}00_{11}19_{6}00_{0}$ C

no action taken

These operations do nothing.

The M-unit NOP is encoded as a UNS_LOADQ into register r0 from register r0. The A-unit NOP is encoded as an INT_IMM into register r0. The C-unit NOP is encoded as a CLOCK into register r0.

RAISES

(nothing)

COUNTS AS

M-unit NOP as CNT_M_NOP; A-unit NOP as CNT_A_NOP; C-unit NOP as CNT_C_NOP

(**PROBE_DISP** *r s lev access disp*) $\quad {}_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}lev_{19}access_{18}0_{16}sdisp_{5}11_{0}$ **MC**

```
maplevel ← level required for $'access$ at $s + 'disp$;
if ($LEVEL < maplevel$) and ($'lev > LEVEL$) then
    raise data protection level exception
else if ($min('lev, LEVEL) \geq maplevel$) and ($s$ has proper access control) then
    $r$ ← pointer to the last byte in the segment
else
    $r \leftarrow 0$
end
```

{**where** $'disp \in [0 \ldots 16383]$, $'sdisp = 'disp/8$}

(**PROBE_INDEX** *r s lev access y*) $\quad {}_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}lev_{19}access_{18}0_{16}y_{11}00_{6}39_{0}$ **MC**

```
maplevel ← level required for $'access$ at $s + 8 * y$;
if ($LEVEL < maplevel$) and ($'lev > LEVEL$) then
    raise data protection level exception
else if ($min('lev, LEVEL) \geq maplevel$) and ($s$ has proper access control) then
    $r$ ← pointer to the last byte in the segment
else
    $r \leftarrow 0$
end
```

These operations are intended for checking the validity of address parameters passed from routines at lower protection levels to routines at higher protection levels. The protection level to check privilege is given by *'lev*. It is specified by a member of the Level enumeration, such as LEV_USER (see §8.1). If the pointer *s* lies beyond the map limit for this domain, the map level is set to LEV_IPL. The kind of access check given by *'access* is one of the following codes:

Name	Value	Meaning
ProbeControl		
P_READ	0	check if the address is mapped for reading
P_MODIFY	1	check if the address is mapped for modification

Besides checking whether the addressed location can be read or written at the given privilege level, these operations make sure that forwarding, trapping, and memory full bit testing are all disabled in the pointer *s*. The pointer returned in *r* has the same access control field as *s*.

RAISES

data_prot

SEE ALSO

DATA_MAP_

ProbeControl

(PROGRAM_CACHE_FLUSH u) $\;{}_{64}\cdots{}_{47}0{}_{44}0{}_{42}u{}_{37}00{}_{32}04{}_{27}0A{}_{21}\cdots{}_{0}$ A

flush (program instruction at u) from program instruction cache

(PROGRAM_CACHE_FLUSH_ANY u) $\;{}_{64}\cdots{}_{47}0{}_{44}0{}_{42}u{}_{37}00{}_{32}07{}_{27}0A{}_{21}\cdots{}_{0}$ A

flush any (program instruction at u) from program instruction cache

(PROGRAM_CACHE_FLUSH_L1 u) $\;{}_{64}\cdots{}_{47}0{}_{44}0{}_{42}u{}_{37}00{}_{32}05{}_{27}0A{}_{21}\cdots{}_{0}$ A

flush any (program instruction at u) from L1 program instruction cache

These are supervisor-privileged operations to maintain consistency in the program instruction caches.

The address in u is a physical word offset into local memory. The PROGRAM_CACHE_FLUSH operation is used to flush a single page of instructions, as after storing new data in a page frame. The PROGRAM_CACHE_FLUSH_ANY operation is used to flush any entries from the caches, presumably only during system initialization. Since the cache is not fully associative, multiple flushes may be required—each flush should specify a different cache line. See §7.2. The PROGRAM_CACHE_FLUSH_L1 operation flushes the L1 cache only. This restriction allows multiple pages to be flushed from L1 more quickly without disturbing the L2 cache.

RAISES

privileged

SEE ALSO

PROGRAM_MAP_

(PROGRAM_MAP_FLUSH *u*) $_{64}\cdots_{47}0_{44}0_{42}u_{37}00_{32}02_{27}0A_{21}\cdots_{0}$ **A**

flush (program map at *u*) from program map cache

(PROGRAM_MAP_FLUSH_ANY *u*) $_{64}\cdots_{47}0_{44}0_{42}u_{37}00_{32}03_{27}0A_{21}\cdots_{0}$ **A**

flush any (program map at *u*) from program map cache

These are supervisor-privileged operations to maintain consistency in the program address translation cache.

The Bits 63–60 and Bits 31–12 of *u* address the program map cache entry; other bits of *u* are ignored. A violation of the map limit will not raise a map limit exception. The PROGRAM_-MAP_FLUSH operation is used to flush a single map entry, as after changing the program map in I/O memory. The PROGRAM_MAP_FLUSH_ANY operation is used to flush any map entry for the given domain from the cache. Since the cache is not fully associative, up to 128 flushes may be required; each flush should specify a different page modulo 128. See §7.1.

RAISES

privileged

SEE ALSO

DATA_MAP_

PROGRAM_MAP_

(PROGRAM_STATE_RESTORE *u*)

```
   . . . 0  0  u  00  00  0A . . .         A
64     47 44 42 37   32  27   21   0
```

(program state descriptor for domain in $u\_63 - 60$) — u

This supervisor-privileged operation is used to set the program state descriptor.

The register u contains a program state descriptor: see §8.5. The Bits 63-60 of u specify the protection domain, i.e. the program state descriptor is self-tagged.

RAISES

privileged

SEE ALSO

DATA_STATE_

(PTR_SET_AC *t u ac*) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}\mathrm{1A}_{32}ac_{27}\mathrm{08}_{21}\cdots{}_{0}$ **A**

$t \leftarrow u$, with $'ac$

(PTR_SET_AC *x y ac*) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}ac_{6}\mathrm{20}_{0}$ **C**

$x \leftarrow y$, with $'ac$

These operations modify the access control field of pointers. That is. if the resulting pointer $(t \vee x)$ is used without accompanying access control field in a subsequent memory reference operation, then the effect will be as if $u \vee y$ and ac had been used instead.

RAISES

(nothing)

SEE ALSO

LOAD_FE.STORE

PTR_SET_

(REAL_FLOAT *t* *v*) $_{64}\ldots{}_{47}\,t\,{}_{42}\,00\,{}_{37}\,v\,{}_{32}\,19\,{}_{27}\,0E\,{}_{21}\ldots{}_{0}$ A

t — rounded float *v*

This operation is used to do rounding conversions from 64-bit floating point to 32-bit floating point. The inverse of this operation is FLOAT_REAL.

If *v* is NaN, *t* is that NaN after rounding the least significant bits from the significand and or'ing in a one at bit 3 to preserve NaN identity.

RAISES

float_overflow, float_underflow, float_inexact

SEE ALSO

FLOAT_REAL

(REG_LOAD_AC_DISP *r s ac disp*) $\quad {}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{D}_{47}\cdots{}_{21}ac_{16}sdisp_{5}01_{0}$ **MC**

$r \leftarrow$ (word at $s +$ ’*disp*), with ’*ac*;
(poison at ’*r*) $\leftarrow$ (full at $s +$ ’*disp*)
{**where** ’*disp* $\in [0 \ldots 16383]$, ’*sdisp* = ’*disp*/8}

(REG_LOAD_AC_INDEX *r s ac y*) $\quad {}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{F}_{47}\cdots{}_{21}ac_{16}y_{11}09_{6}39_{0}$ **MC**

$r \leftarrow$ (word at $s + 8 * y$), with ’*ac*;
(poison at ’*r*) $\leftarrow$ (full at $s + 8 * y$)

(REG_LOAD_DISP *r s disp*) $\quad {}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{D}_{47}\cdots{}_{21}sdisp_{5}00_{0}$ **MC**

$r \leftarrow$ (word at $s +$ ’*disp*)
(poison at ’*r*) $\leftarrow$ (full at $s +$ ’*disp*)
{**where** ’*disp* $\in [0 \ldots 524287]$, ’*sdisp* = ’*disp*/8}

(REG_LOAD_INDEX *r s y*) $\quad {}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{F}_{47}\cdots{}_{21}00_{16}y_{11}09_{6}38_{0}$ **MC**

$r \leftarrow$ (word at $s + 8 * y$)
(poison at ’*r*) $\leftarrow$ (full at $s + 8 * y$)

These operations load the word and the memory full bit of the access state (§6.1) from the addressed memory cell. The word is stored in register *r* and the memory full bit is stored in the poison bit for register *r*.

These operations are only subject to the trapping and forwarding normally controlled by the access state of the addressed memory location. If *ac* is present, its forward, data trap0 and data trap1 disable bits are used; otherwise those of *s* are used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_LOAD

SEE ALSO

REG_STORE, REG_MOVE

(REG_MOVE t v) $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,09\,{}_{37}\,v\,{}_{32}\,00\,{}_{27}\,00\,{}_{21}\cdots{}_{0}$ A

$t \leftarrow v$;
(poison at 't) ← (poison at 'v)

(REG_MOVE x y) $\quad {}_{64}\cdots{}_{21}\,x\,{}_{16}\,y\,{}_{11}\,02\,{}_{6}\,00\,{}_{0}$ C

$x \leftarrow y$;
(poison at 'x) ← (poison at 'y)

These operations copy data from one register to another, without raising a poison exception.

RAISES
(nothing)

(REG_STORE_AC_DISP *r s ac disp*) $_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}ac_{16}sdisp_{5}01_{0}$ MC

(word at $s +$ *'disp*) ← r. with *'ac*:
(full at $s +$ *'disp*) ← (poison at *'r*)
{where *'disp* $\in [0 \ldots 16383]$, *'sdisp* = *'disp*/8}

(REG_STORE_AC_INDEX *r s ac y*) $_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}ac_{16}y_{11}01_{6}39_{0}$ MC

(word at $s + 8 * y$) ← r, with *'ac*;
(full at $s + 8 * y$) ← (poison at *'r*)

(REG_STORE_DISP *r s disp*) $_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}sdisp_{5}00_{0}$ MC

(word at $s +$ *'disp*) ← r
(full at $s +$ *'disp*) ← (poison at *'r*)
{where *'disp* $\in [0 \ldots 524287]$, *'sdisp* = *'disp*/8}

(REG_STORE_INDEX *r s y*) $_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}00_{16}y_{11}01_{6}38_{0}$ MC

(word at $s + 8 * y$) ← r
(full at $s + 8 * y$) ← (poison at *'r*)

These operations store both the memory full bit of the access state(§6.1) and the value in the addressed memory cell. The value comes from register r. The memory full bit comes from the poison bit associated with register r, complementing the action of a REG_LOAD operation. Thus, these operations are not subject to a poison exception due to r.

These operations are only subject to the trapping or forwarding normally controlled by the access state of the addressed memory location. If *ac* is present, its forward, data trap0 and data trap1 disable bits are used; otherwise those of s are used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_STORE

SEE ALSO

REG_LOAD, REG_MOVE

REG_STORE_

(RESULTCODE_SAVE x) $_{64}\ldots{}_{21}\,x\,{}_{16}\,00\,{}_{11}\,1D\,{}_{6}\,00\,{}_{0}$ C

x — RESULTCODE ← lookahead index
RESULTCODE — 0

This operation saves and clears the result code register, described in §9.1. The value of RESULTCODE is undefined after an instruction combining RESULTCODE_SAVE with a floating-point A-operation. The data resultcodes are rotated so that *dr*0 corresponds to *opa*0.

RAISES
(nothing)

SEE ALSO
EXCEPTION_, DATA_OPA_SAVE

(ROTATE_LEFT x y z) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}z_{6}\,\mathrm{1A}_{0}$ C

$x \leftarrow y \hookleftarrow z$

(ROTATE_LEFT_TEST x y z) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}z_{6}\,\mathrm{1B}_{0}$ C

$x \leftarrow y \hookleftarrow z$

(ROTATE_RIGHT x y z) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}z_{6}\,02_{0}$ C

$x \leftarrow y \hookrightarrow z$

(ROTATE_RIGHT_TEST x y z) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}z_{6}\,03_{0}$ C

$x \leftarrow y \hookrightarrow z$

These operations rotate a word to the left or right. They compute the unsigned rotation amount z modulo 64.

The _TEST version generates carry if a 1-bit is rotated out of one end of y and into the other end, and never generates overflow/NaN.

RAISES

(nothing)

SEE ALSO

SHIFT_LEFT, INT_SHIFT_RIGHT, UNS_SHIFT_RIGHT, SHIFT_PAIR

(SELECT_FLOAT *t u v floatselect cn*) $_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathit{floatselect}_{29}cn_{27}06_{21}\ldots{}_{0}$ A

$t \leftarrow$ if $cv_{cn} \in$ '*floatselect* then u else v end

(SELECT_FLOAT_TEST *t u v floatselect cn*) $_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathit{floatselect}_{29}cn_{27}07_{21}\ldots{}_{0}$ A

$t \leftarrow$ if $cv_{cn} \in$ '*floatselect* then u else v end

(SELECT_INT *t u v intselect cn*) $_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathit{intselect}_{29}cn_{27}04_{21}\ldots{}_{0}$ A

$t \leftarrow$ if $cv_{cn} \in$ '*intselect* then u else v end

(SELECT_INT_TEST *t u v intselect cn*) $_{64}\ldots{}_{47}t_{42}u_{37}v_{32}\mathit{intselect}_{29}cn_{27}05_{21}\ldots{}_{0}$ A

$t \leftarrow$ if $cv_{cn} \in$ '*intselect* then u else v end

These operations are used to conditionally select the value u or v.

The masks *intselect* and *floatselect* describe the values of the condition in cv_{cn} that will select u rather than v: see §4.

The condition test *intselect* may be one of SEL_CY, SEL_EQ, SEL_IGT, SEL_IGE, SEL_UGT, SEL_UGE, SEL_IPL, or SEL_IPZ. Reversal of u and v effectively yields the additional tests SEL_NC, SEL_NE, SEL_ILE, SEL_ILT, SEL_ULE, SEL_ULT, SEL_IMZ, and SEL_IMI. The condition test *floatselect* may be one of SEL_FLT, SEL_FGE, SEL_FGT, SEL_FLE, or SEL_FUN. Reversing u and v yields additional nameless conditions. Selection based on floating-point equality may use a SELECT_INT operation with SEL_EQ.

These operations are "lazy" in that Poison in the selected value is merely propagated to the destination. Hence, no exceptions are raised if either value is poisoned.

The _TEST versions of these operations never generate overflow/NaN or carry.

RAISES

(nothing)

SEE ALSO

INT_MAX, INT_MIN, FLOAT_MAX, FLOAT_MIN, BIT_MERGE

(SHIFT_LEFT *x y z*) $_{64}\cdots{}_{21}x_{16}y_{11}z_{6}18_{0}$ **C**

$x — y \ll z$

(SHIFT_LEFT_TEST *x y z*) $_{64}\cdots{}_{21}x_{16}y_{11}z_{6}19_{0}$ **C**

$x — y \ll z$

(SHIFT_LEFT_IMM *t u sh*) $_{64}\cdots{}_{47}t_{42}u_{37}8_{33}sh_{27}08_{21}\cdots{}_{0}$ **A**

$t — u \ll 'sh$
{**where** $'sh \in [0 \ldots 63]$}

(SHIFT_LEFT_IMM_TEST *t u sh*) $_{64}\cdots{}_{47}t_{42}u_{37}8_{33}sh_{27}09_{21}\cdots{}_{0}$ **A**

$t — u \ll 'sh$
{**where** $'sh \in [0 \ldots 63]$}

These operations shift words to the left, filling vacated positions on the right with 0-bits. Unsigned shift counts in *z* are taken modulo 64.

The _TEST version generates carry if a 1-bit is shifted out of *u* or *y*, and never generates overflow/NaN.

RAISES

(nothing)

SEE ALSO

UNS_SHIFT_RIGHT, INT_SHIFT_RIGHT, SHIFT_PAIR, ROTATE_LEFT

SHIFT_LEFT_

(**SHIFT_PAIR_LEFT** $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}14_{21}\cdots{}_{0}$ **A**

t — (the pair $(u, v) \ll w)/2^{64}$

(**SHIFT_PAIR_LEFT_TEST** $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}15_{21}\cdots{}_{0}$ **A**

t — (the pair $(u, v) \ll w)/2^{64}$

(**SHIFT_PAIR_RIGHT** $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}16_{21}\cdots{}_{0}$ **A**

t — (the pair $(u, v) \gg w) mod 2^{64}$

(**SHIFT_PAIR_RIGHT_TEST** $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}t_{42}u_{37}v_{32}w_{27}17_{21}\cdots{}_{0}$ **A**

t — (the pair $(u, v) \gg w) mod 2^{64}$

SHIFT_PAIR_LEFT shifts a copy of u left and fills vacated positions with bits from the left end of v, whereas SHIFT_PAIR_RIGHT shifts a copy of v right and fills vacated positions with bits from the right end of u. Unsigned shift counts in w are taken modulo 64.

The SHIFT_PAIR_LEFT_TEST version generates carry if those bits of u not appearing in t are not all 0, and never generates overflow/NaN. The SHIFT_PAIR_RIGHT_TEST version generates carry if those bits of v not appearing in t are not all 0, and never generates overflow/NaN.

RAISES

(nothing)

SEE ALSO

SHIFT_LEFT, ROTATE_LEFT, UNS_SHIFT_RIGHT, INT_SHIFT_RIGHT

(SKIP *mask cn offset*) $_{64}\ldots{}_{21}\,mask\,_{13}\,cn\,_{11}\,hi\,_{7}\,1\,_{6}\,6\,_{3}\,lo\,_{0}$ C

```
if $cv_{cn} \in$ '$mask$ then
    ssw.pc $\leftarrow$ ssw.pc + ( '$offset$ + 1)
end
  {where '$offset \in [0 \ldots 119]$, '$lo$ = '$offset$ mod 8, '$hi = \lfloor$'$offset/8\rfloor$}
```

(SKIP_OFTEN *mask cn offset*) $_{64}\ldots{}_{21}\,mask\,_{13}\,cn\,_{11}\,hi\,_{7}\,0\,_{6}\,7\,_{3}\,lo\,_{0}$ C

```
if $cv_{cn} \in$ '$mask$ then
    ssw.pc $\leftarrow$ ssw.pc + ( '$offset$ + 1)
else
    stop lookahead
end
  {where '$offset \in [0 \ldots 119]$, '$lo$ = '$offset$ mod 8, '$hi = \lfloor$'$offset/8\rfloor$}
```

(SKIP_SELDOM *mask cn offset*) $_{64}\ldots{}_{21}\,mask\,_{13}\,cn\,_{11}\,hi\,_{7}\,1\,_{6}\,7\,_{3}\,lo\,_{0}$ C

```
if $cv_{cn} \in$ '$mask$ then
    ssw.pc $\leftarrow$ ssw.pc + ( '$offset$ + 1)
    stop lookahead
end
  {where '$offset \in [0 \ldots 119]$, '$lo$ = '$offset$ mod 8, '$hi = \lfloor$'$offset/8\rfloor$}
```

These operations do conditional forward branches if the selected condition code is a member of the condition *mask*. If taken, the skip skips the following *offset* instructions. The value *mask* is an eight-bit wide condition mask of type CondMask, described in §4. If *mask* is empty, then the skip always fails.

SKIP_OFTEN and SKIP_SELDOM can be monitored via CNT_JUMP_EXPECTED and CNT_-JUMP_UNEXPECTED to observe branch prediction accuracy. SKIP only counts toward CNT_-TRANSFER_TOTAL.

RAISES

(nothing)

COUNTS AS

CNT_JUMP_EXPECTED if expected path taken, CNT_JUMP_UNEXPECTED if unexpected path taken, CNT_TRANSFER_TOTAL

SEE ALSO

JUMP

SKIP_

(SSW_DISP *x* *offset*)

```
...  x   0   0   hi  0  6  lo      C
64   21  16  13  11  7  6  3  0
```

$x \leftarrow$ SSW + $'offset + 1$

{**where** $'offset \in [0 \ldots 119]$. $'lo = 'offset \bmod 8$. $'hi = \lfloor 'offset/8 \rfloor$}

(SSW_RESTORE *u*)

```
...  0   0   u   00  0A  0A  ...     A
64   47  44  42  37  32  27  21  0
```

SSW.*cv* ← *u.cv*
SSW.*tm* ← *u.tm*
SSW.*md* ← *u.md*

The SSW_DISP operation is used to load a branch address into a general purpose register *x*, rather than a target register. Thus, the value may be later loaded into a target with TARGET_RESTORE prior to jumping to the location. It also returns the trap, mode, and condition fields of the SSW.

The SSW_RESTORE operation is used to set the trap, mode, and condition fields in the SSW. The value of CV after an instruction combining SSW_RESTORE with a _TEST C-op is undefined.

RAISES

(nothing)

SEE ALSO

TARGET_RESTORE

(STATE_LOAD_DISP *r s disp*) $_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}sdisp_{5}10_{0}$ MC

r — (access state at $s + {}'disp$)
{where $'disp \in [0 \ldots 524287]$, $'sdisp = {}'disp/8$}

(STATE_LOAD_INDEX *r s y*) $_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}00_{16}y_{11}00_{6}38_{0}$ MC

r — (access state at $s + 8 * y$)

These operations load the access state(§6.1) from the addressed memory cell, and convert the access state into an access control field(§6.1) stored as a pointer in register *r*. Word alignment is not required; the byte select bits are ignored.

These operations are not subject to the trapping, forwarding, or memory full bit waiting normally controlled by the access state of the addressed memory location.

The pointer *r* is constructed as follows (the access control field is constructed by inverting the operations done by a STATE_STORE operation):

- A copy of the forward enable bit (field "forward_enable" in the access state) is placed in field "fwd_disable" of *r*.
- A copy of data trap bit 0 (field "trap0_enable" in the access state) is placed in both field "trap0_store_disable" and field "trap0_load_disable" of *r*.
- A copy of data trap bit 1 (field "trap1_enable" in the access state) is placed in both field "trap1_store_disable" and field "trap1_load_disable" of *r*.
- The memory full bit (field "full" in the access state) is used to construct the full/empty control (field "fe_control"). That field is set to FE_FUTURE if the memory full bit is true, and is set to FE_SYNC if the memory full bit is false.
- Other bits of *r* are set to 0.

RAISES

data_hw_error. data_prot

COUNTS AS

CNT_LOAD

SEE ALSO

STATE_STORE. STATE_LOCK

STATE_LOAD_

(STATE_LOCK_AC_DISP *r s ac disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{C}{}_{47}\cdots{}_{21}ac{}_{16}sdisp{}_{3}01{}_{0}$ MC

r ← (access state at s + '$disp$), with 'ac;
(access state at s + '$disp$) ← (forwarded, empty)
{**where** '$disp \in [0 \ldots 16383]$, '$sdisp$ = '$disp/8$}

(STATE_LOCK_AC_INDEX *r s ac y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}ac{}_{16}y{}_{11}11{}_{6}39{}_{0}$ MC

r ← (access state at $s + 8 * y$), with 'ac;
(access state at $s + 8 * y$) ← (forwarded, empty)

(STATE_LOCK_DISP *r s disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{C}{}_{47}\cdots{}_{21}sdisp{}_{3}00{}_{0}$ MC

r ← (access state at s + '$disp$);
(access state at s + '$disp$) ← (forwarded, empty)
{**where** '$disp \in [0 \ldots 524287]$, '$sdisp$ = '$disp/8$}

(STATE_LOCK_INDEX *r s y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}00{}_{16}y{}_{11}11{}_{6}38{}_{0}$ MC

r ← (access state at $s + 8 * y$);
(access state at $s + 8 * y$) ← (forwarded, empty)

These operations allow atomic access state manipulation, with additional access control modification through the operand *ac*. The operation loads the access state(§6.1) from the addressed memory cell, and converts the access state into an access control field(§6.1) stored as a pointer in register *r*, as is done by a STATE_LOAD operation. The operation then sets the access state stored in the addressed memory cell to forwarded and empty. Word alignment is not required: the byte select bits are ignored.

These operations are not subject to the forwarding or memory full bit waiting normally controlled by the access state of the addressed memory location, except that a location that is both forwarded and not full is considered locked. In this case, the operation fails and is retried later. However, data trap bits are observed as in a normal memory operation. If *ac* is present, its data trap0 and data trap1 disable bits are used; otherwise those of *s* are used.

The STATE_LOCK operation allows a possibly unforwarded memory word to be forwarded in an indivisible manner, locking the word with an "empty forwarding pointer" access state, while retrieving its current value.

RAISES

data_hw_error, data_prot

COUNTS AS

CNT_STORE

SEE ALSO

STATE_LOAD, STATE_STORE

(STATE_SCRUB_DISP *r s disp*) $${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}sdisp{}_{5}08{}_{0}$$ **MC**

```
    $dsyn \leftarrow$ (data syndrome at $s + {}'disp$)
    $asyn \leftarrow$ (access syndrome at $s + {}'disp$)
    $data \leftarrow$ (word at $s + {}'disp$)
    for $i \in [0 \ldots 7] : r_i \leftarrow dsyn_i$
    for $i \in [8 \ldots 11] : r_i \leftarrow asyn_{i-8}$
    for $i \in [12 \ldots 63] : r_i \leftarrow data_i$
        {where $'disp \in [0 \ldots 524287]$, $'sdisp = {}'disp/8$}
```

(STATE_SCRUB_INDEX *r s y*) $${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}00{}_{16}y{}_{11}02{}_{6}38{}_{0}$$ **MC**

```
    $dsyn \leftarrow$ (data syndrome at $s + 8 * y$)
    $asyn \leftarrow$ (access syndrome at $s + 8 * y$)
    $data \leftarrow$ (word at $s + 8 * y$)
    for $i \in [0 \ldots 7] : r_i \leftarrow dsyn_i$
    for $i \in [8 \ldots 11] : r_i \leftarrow asyn_{i-8}$
    for $i \in [12 \ldots 63] : r_i \leftarrow data_i$
```

These operations atomically load and store the access state (§6.1) and load the data of the addressed memory cell to correct single-bit errors before they become uncorrectable multiple bit errors. Multiple-bit errors must be detected by examination of the syndrome bits. The combined syndrome bits for the data word and access state are returned. If there are no errors detected by the error-correction control logic, then these bits will be zero.

The appendix details the syndrome values returned.

These operations are not subject to the trapping, forwarding, or memory full bit waiting normally controlled by the access state of the addressed memory location.

RAISES

data_hw_error, data_prot. data_alignment, data_blocked

STATE_SCRUB_

(STATE_STORE_AC_DISP *r s ac disp*) ${}_{64}\cdots{}_{61}r_{56}s_{51}E_{47}\cdots{}_{21}ac_{16}sdisp_{5}01_{0}$ MC

(word at $s +$ '*disp*) — *r*;
(access state at $s +$ '*disp*) — accesscontrol(s), with '*ac*
{where '*disp* $\in [0 \ldots 16383]$, '*sdisp* = '*disp*/8}

(STATE_STORE_AC_INDEX *r s ac y*) ${}_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}ac_{16}y_{11}19_{6}39_{0}$ MC

(word at $s + 8 * y$) — *r*;
(access state at $s + 8 * y$) — accesscontrol(s), with '*ac*

(STATE_STORE_DISP *r s disp*) ${}_{64}\cdots{}_{61}r_{56}s_{51}E_{47}\cdots{}_{21}sdisp_{5}00_{0}$ MC

(word at $s +$ '*disp*) — *r*;
(access state at $s +$ '*disp*) — accesscontrol(s)
{where '*disp* $\in [0 \ldots 524287]$, '*sdisp* = '*disp*/8}

(STATE_STORE_INDEX *r s y*) ${}_{64}\cdots{}_{61}r_{56}s_{51}F_{47}\cdots{}_{21}00_{16}y_{11}19_{6}38_{0}$ MC

(word at $s + 8 * y$) — *r*;
(access state at $s + 8 * y$) — accesscontrol(s)

These operations store both the access state(§6.1) and the value in the addressed memory cell. The value comes from register *r*. The access state comes from the access control field(§6.1) of register *s*, inverting the encoding done by a STATE_LOAD operation.

These operations are not subject to the trapping, forwarding, or memory full bit waiting normally controlled by the access state of the addressed memory location.

The access state in the memory cell is constructed from the access control field of *s* as follows:

- A copy of the forward disable bit (field "fwd_disable" in the access control) is placed in the forward enable bit, field "forward_enable", in the access state.
- The logical OR of field "trap0_store_disable" and field "trap0_load_disable" is placed in the "data trap 0" enable bit, field "trap0_enable", in the access state.
- The logical OR of field "trap1_store_disable" and field "trap1_load_disable" is placed in the "data trap 1" enable bit, field "trap1_enable", in the access state.
- The memory full bit, field "full", is set if the full/empty control (field "fe_control") is set to FE_FUTURE; with FE_SYNC, the memory full bit is cleared. The result is undefined if the field is set to FE_NORMAL.

The exception is that if the *ac* (access control) operand is present, then the forwarding and data trap disable bits and full/empty control bits from *ac* replace those from *s*.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_STORE

SEE ALSO

STATE_LOAD, STATE_LOCK

(STATE_STORE_ERROR_DISP *r s disp*) $_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{E}_{47}\cdots{}_{21}00_{16}\mathit{sdisp}_{5}01_{0}$ MC

(word at $s +$ '*disp*) — *r*;
(access state at $s +$ '*disp*) — corrected access state
{**where** '*disp* $\in [0 \ldots 16383]$, '*sdisp* = '*disp*/8}

(STATE_STORE_ERROR_INDEX *r s y*) $_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{F}_{47}\cdots{}_{21}0_{16}y_{11}19_{6}39_{0}$ MC

(word at $s + 8 * y$) — *r*;
(access state at $s + 8 * y$) — corrected access state

These operations store the value and correct the access state(§6.1) in the addressed memory cell, as long as there is a correctable error in the old value stored. The value comes from register *r*.

These operations are not subject to the trapping, forwarding, or memory full bit waiting normally controlled by the access state of the addressed memory location.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_STORE

SEE ALSO

STATE_SCRUB

STATE_STORE_ERROR_

(STOREB *r s*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{B}{}_{47}\cdots{}_{0}$ M

(byte at *s*) ← *r*

(STOREB_AC_DISP *r s ac disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{E}{}_{47}\cdots{}_{21}ac{}_{16}disp{}_{2}3{}_{0}$ MC

(byte at *s* + '*disp mod* 2^{48}) ← *r*, with '*ac*
{where '*disp* ∈ [0 . . . 16383]}

(STOREB_AC_INDEX *r s ac y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}ac{}_{16}y{}_{11}\mathrm{1E}{}_{6}39{}_{0}$ MC

(byte at *s* + *y mod* 2^{48}) ← *r*, with '*ac*

(STOREB_DISP *r s disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{E}{}_{47}\cdots{}_{21}disp{}_{2}2{}_{0}$ MC

(byte at *s* + '*disp mod* 2^{48}) ← *r*
{where '*disp* ∈ [0 . . . 524287]}

(STOREB_INDEX *r s y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}00{}_{16}y{}_{11}\mathrm{1E}{}_{6}38{}_{0}$ MC

(byte at *s* + *y mod* 2^{48}) ← *r*

These operations store a byte at the addressed location. If *ac* is present, it is used; otherwise the access control field of *s* is used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_STORE

SEE ALSO

INT_LOADB, UNS_LOADB

(STOREH *r s*) ${}_{64}\cdots{}_{61}r_{56}s_{51}9_{47}\cdots{}_{0}$ **M**

(halfword at s) $\leftarrow r$

(STOREH_AC_DISP *r s ac disp*) ${}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{E}_{47}\cdots{}_{21}ac_{16}sdisp_{4}9_{0}$ **MC**

(halfword at $s +$ *'disp mod* 2^{48}) $\leftarrow r$, with *'ac*
{where *'disp* $\in [0\ldots16383]$, *'sdisp* = *'disp*/4}

(STOREH_AC_INDEX *r s ac y*) ${}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{F}_{47}\cdots{}_{21}ac_{16}y_{11}\mathrm{1A}_{6}39_{0}$ **MC**

(halfword at $s + 4 * y$ *mod* 2^{48}) $\leftarrow r$, with *'ac*

(STOREH_DISP *r s disp*) ${}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{E}_{47}\cdots{}_{21}sdisp_{4}8_{0}$ **MC**

(halfword at $s +$ *'disp mod* 2^{48}) $\leftarrow r$
{where *'disp* $\in [0\ldots524287]$, *'sdisp* = *'disp*/4}

(STOREH_INDEX *r s y*) ${}_{64}\cdots{}_{61}r_{56}s_{51}\mathrm{F}_{47}\cdots{}_{21}00_{16}y_{11}\mathrm{1A}_{6}38_{0}$ **MC**

(halfword at $s + 4 * y$ *mod* 2^{48}) $\leftarrow r$

These operations store a halfword at the addressed location. If *ac* is present, it is used; otherwise the access control field of *s* is used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_STORE

SEE ALSO

INT_LOADH, UNS_LOADH

STOREH_

(STOREQ *r s*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{A}{}_{47}\cdots{}_{0}$ M

(quarterword at *s*) — *r*

(STOREQ_AC_DISP *r s ac disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{E}{}_{47}\cdots{}_{21}ac{}_{16}sdisp{}_{3}5{}_{0}$ MC

(quarterword at $s +$ '*disp mod* 2^{46}) — *r*, with '*ac*
{**where** '*disp* $\in [0 \ldots 16383]$, '*sdisp* = '*disp*/2}

(STOREQ_AC_INDEX *r s ac y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}ac{}_{16}y{}_{11}\mathrm{1C}{}_{6}39{}_{0}$ MC

(quarterword at $s + 2 * y$ *mod* 2^{48}) — *r*, with '*ac*

(STOREQ_DISP *r s disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{E}{}_{47}\cdots{}_{21}sdisp{}_{3}4{}_{0}$ MC

(quarterword at $s +$ '*disp mod* 2^{48}) — *r*
{**where** '*disp* $\in [0 \ldots 524287]$, '*sdisp* = '*disp*/2}

(STOREQ_INDEX *r s y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}00{}_{16}y{}_{11}\mathrm{1C}{}_{6}38{}_{0}$ MC

(quarterword at $s + 2 * y$ *mod* 2^{48}) — *r*

These operations store a quarterword at the addressed location. If *ac* is present, it is used; otherwise the access control field of *s* is used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_STORE

SEE ALSO

INT_LOADQ, UNS_LOADQ

(STORE *r s*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}8{}_{47}\cdots{}_{0}$ M

(word at *s*) $\leftarrow r$

(STORE_AC_DISP *r s ac disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{E}{}_{47}\cdots{}_{21}ac{}_{16}sdisp{}_{5}11{}_{0}$ MC

(word at $s +$ *'disp mod* 2^{48}) $\leftarrow r$, with *'ac*
{where *'disp* $\in [0 \ldots 16383]$, *'sdisp* = *'disp*/8}

(STORE_AC_INDEX *r s ac y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}ac{}_{16}y{}_{11}18{}_{6}39{}_{0}$ MC

(word at $s + 8 * y$ *mod* 2^{48}) $\leftarrow r$, with *'ac*

(STORE_DISP *r s disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{E}{}_{47}\cdots{}_{21}sdisp{}_{5}10{}_{0}$ MC

(word at $s +$ *'disp mod* 2^{48}) $\leftarrow r$
{where *'disp* $\in [0 \ldots 524287]$, *'sdisp* = *'disp*/8}

(STORE_INDEX *r s y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}00{}_{16}y{}_{11}18{}_{6}38{}_{0}$ MC

(word at $s + 8 * y$ *mod* 2^{48}) $\leftarrow r$

These operations store a word at the addressed location. If *ac* is present, it is used; otherwise the access control field of *s* is used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_STORE

SEE ALSO

LOAD, STOREB

STORE_

(STREAM_CUR_SAVE t) $\quad {}_{64}\cdots{}_{47}t{}_{42}\,08\,{}_{37}00\,{}_{32}02\,{}_{27}00\,{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow \mathrm{SCUR}_D$

(STREAM_IDENTIFIER_SAVE t) $\quad {}_{64}\cdots{}_{47}t{}_{42}\,08\,{}_{37}00\,{}_{32}01\,{}_{27}00\,{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow$ identifier of the executing stream

(STREAM_LOOKAHEAD_SAVE x) $\quad {}_{64}\cdots{}_{21}x{}_{16}\,00\,{}_{11}1F\,{}_{6}00\,{}_{0}$ **C**

$x \leftarrow$ (lookahead index of the executing stream) $* 4$

(STREAM_RES_SAVE t) $\quad {}_{64}\cdots{}_{47}t{}_{42}\,08\,{}_{37}00\,{}_{32}03\,{}_{27}00\,{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow \mathrm{SRES}_D$

The STREAM_CUR_SAVE and STREAM_RES_SAVE operations respectively return the number of streams currently executing and the number of streams currently reserved in the stream's protection domain.

The STREAM_IDENTIFIER_SAVE operation is meant to help diagnose the stream management hardware. STREAM_IDENTIFIER_SAVE returns the issuing stream's stream number that was assigned by the hardware when the stream was created.

The STREAM_LOOKAHEAD_SAVE operation is used to read the three-bit lookahead lock counter index. It is used for mapping between the OPA and OPD registers of the M-unit and the data result codes. Since the data result codes are four-bit values, the index returned is scaled by four.

RAISES

(nothing)

SEE ALSO

STREAM_RESERVE, STREAM_CREATE, STREAM_QUIT, §2

(STREAM_CATCH *r t x delay str*)

$_{64}\cdots{}_{61}\,r\,_{56}\,00\,_{51}\,F\,_{47}\,t\,_{42}\,06\,_{37}\,00\,_{32}\,delay\,_{30}\,0\,_{27}\,00\,_{21}\,x\,_{16}\,0\,_{15}\,str\,_{8}\,delay\,_{6}\,0C\,_{0}$ MAC

```
SSW.pc — newpc;
SSW.md — newmd;
SSW.tm — newtm;
SSW.cv — 0;
D ← newdomain;
LEVEL — newlevel;
r — data;
t ← data;
x ← data;
T0 — data;
EXCEPTION is cleared;
RESULTCODE is cleared;
instruction counter — 0;
```

This operation is internally generated by the hardware to complete the execution of a STREAM_-CREATE instruction. As such, IPL privilege is required to explicitly execute it.

RAISES

(nothing)

SEE ALSO

STREAM_CREATE

STREAM_CATCH_

(STREAM_COUNT_INST x) $_{64}\ldots{}_{21}\,x\,{}_{16}\,00\,{::}\,18\,{}_{6}\,00\,{}_{0}$ C

x — (instruction counter)

(STREAM_COUNT_INST_RESTORE x y) $_{64}\ldots{}_{21}\,x\,{}_{16}\,y\,{::}\,0A\,{}_{6}\,00\,{}_{0}$ C

x — (instruction counter):
(instruction counter) — y

These operations manipulate the stream's instruction counter. STREAM_COUNT_INST returns the counter. STREAM_COUNT_INST_RESTORE sets the instruction counter, which then counts down, stopping once it hits zero. When the instruction counter steps from one down to zero, the instruction count exception is raised.

There is also an instruction issue counter for each protection domain for accounting and performance measurement.

RAISES

(nothing)

SEE ALSO

COUNT_ISSUES, §10.1

(STREAM_CREATE_IMM *r t u x y offset*) $_{64}\cdots{}_{61}\,r\,{}_{56}\,00\,{}_{51}\,F\,{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,lo\,{}_{27}\,0B\,{}_{21}\,x\,{}_{16}\,y\,{}_{11}\,hi\,{}_{6}\,0E\,{}_{0}$ MAC

```
if SCUR_D = SRES_D then
    raise create exception
else
    SCUR_D ← SCUR_D + 1;
    SSW.pc' ← SSW.pc + 'offset;
    SSW.md' ← SSW.md;
    SSW.tm' ← SSW.tm;
    SSW.cv' ← 0;
    D' ← D;
    LEVEL' ← LEVEL;
    r' ← r;
    t' ← u;
    x' ← y;
    T0' ← T0;
    EXCEPTION' is cleared;
    RESULTCODE' is cleared;
    instruction counter' ← 0;
end
    {where 'offset ∈ [−2^14 ... 2^14 − 1], 'lo = 'offset mod 1024, 'hi = ⌊'offset/1024⌋}
```

This operation attempts to create a new instruction stream. The unprimed registers represent registers in the old stream. The primed registers represent registers in the new stream.

If the current number of streams executing in the protection domain (SCUR_D) is less than the number reserved by prior STREAM_RESERVE_ operations (SRES_D), then the STREAM_CREATE succeeds, and state is copied from the current stream into the new stream.

By convention, target register T0 contains the SSW for the current trap handler. These registers are duplicated in the new stream. The three general-purpose registers r, u, and y that are copied into the new stream will typically be the frame pointer, an argument pointer, and some stream identifier. The general purpose registers other than r, t, and x, the trap registers, and target registers (other than T0) are undefined. The exception register and result code register are cleared (a safe state). The instruction count register is set to zero, which will *not* cause a trap on issues in the stream.

Streams must first be reserved with a STREAM_RESERVE operation, can then created with a STREAM_CREATE operation, and are then killed off with a STREAM_QUIT operation.

RAISES

create

COUNTS AS

CNT_CREATE

SEE ALSO

STREAM_RESERVE, STREAM_QUIT, §12.1

STREAM_CREATE_

(STREAM_QUIT)

```
...  00  00  F   00  04  00  00  00  00  00  00  0A      MAC
64 61  56  51 47  42  37  32  27  21  16  11  6  0
```

```
    $SCUR_D \leftarrow SCUR_D - 1$;
    $SRES_D \leftarrow SRES_D - 1$;
    release trap registers;
    release OPA/OPD slots;
    stop execution;
```

(STREAM_QUIT_PRESERVE)

```
...  00  00  F   00  04  00  01  00  00  00  01  0A      MAC
64 61  56  51 47  42  37  32  27  21  16  11  6  0
```

```
    $SCUR_D \leftarrow SCUR_D - 1$;
    release trap registers;
    release OPA/OPD slots;
    stop execution;
```

The QUIT operation is supervisor-privileged if field "priv_quit" is set in the program state descriptor. Privileged quit mode lets the operating system clear the stream's state before returning it to the hardware for reallocation (possibly to another protection domain).

Lookahead beyond a STREAM_QUIT is forbidden. Streams which allow such lookahead will force the STREAM_QUIT to be retried, wasting issue slots.

The QUIT_PRESERVE operation preserves the reservation for the current stream.

RAISES

privileged

COUNTS AS

CNT_QUIT

SEE ALSO

STREAM_CREATE, STREAM_RESERVE

(STREAM_RESERVE *t u st*)

```
   . . . t  u  0  st  08 . . .      A
64      47 42 37 35 27  21    0
```

```
$req \leftarrow$ if $'st = 0$ then $u$ else $'st$ end;
$t \leftarrow$ if $(req \leq S - \sum_i \text{SRES}_i - limbo)$
 and $(req \leq \text{SLIM}_D - \text{SRES}_D)$
then $req$ else 0 end;
$\text{SRES}_D \leftarrow \text{SRES}_D + t$
   {where $'st \in [0 \ldots 255]$}
```

(STREAM_RESERVE_TEST *t u st*)

```
   . . . t  u  0  st  09 . . .      A
64      47 42 37 35 27  21    0
```

```
$req \leftarrow$ if $'st = 0$ then $u$ else $'st$ end;
$t \leftarrow$ if $(req \leq S - \sum_i \text{SRES}_i - limbo)$
 and $(req \leq \text{SLIM}_D - \text{SRES}_D)$
then $req$ else 0 end;
$\text{SRES}_D \leftarrow \text{SRES}_D + t$
   {where $'st \in [0 \ldots 255]$}
```

(STREAM_RESERVE_UPTO *t u st*)

```
   . . . t  u  1  st  08 . . .      A
64      47 42 37 35 27  21    0
```

```
$req \leftarrow$ if $'st = 0$ then $u$ else $'st$ end;
$t \leftarrow \min(req, S - \sum_i \text{SRES}_i - limbo, \max(\text{SLIM}_D - \text{SRES}_D, 0))$;
$\text{SRES}_D \leftarrow \text{SRES}_D + t$
   {where $'st \in [0 \ldots 255]$}
```

(STREAM_RESERVE_UPTO_TEST *t u st*)

```
   . . . t  u  1  st  09 . . .      A
64      47 42 37 35 27  21    0
```

```
$req \leftarrow$ if $'st = 0$ then $u$ else $'st$ end;
$t \leftarrow \min(req, S - \sum_i \text{SRES}_i - limbo, \max(\text{SLIM}_D - \text{SRES}_D, 0))$;
$\text{SRES}_D \leftarrow \text{SRES}_D + t$
   {where $'st \in [0 \ldots 255]$}
```

These operations are used to reserve streams prior to creating streams with the STREAM_CREATE operation.

The number of streams reserved for the protection domain to which this stream belongs, SRES_D, is incremented by *req* if possible. The result register *t* reflects the amount by which SRES_D was actually changed. The register *u* is an unsigned integer.

The resulting stream reservation will not exceed the larger of the current reservation and SLIM_D, the maximum number of streams allocated to this protection domain. In addition, the sum of all reservations cannot exceed S, the number of streams available in the processor. The operating system may encourage(inhibit) parallelism by setting SLIM_D above(below) SRES_D.

The _TEST versions of these operations never generate overflow/NaN, and generate carry if the reservation SRES_D was changed by exactly *req*.

RAISES

(nothing)

SEE ALSO

STREAM_CREATE, STREAM_QUIT

(TARGET_DISP *tn offset*) $\quad {}_{64}\cdots{}_{47}\,tn\,{}_{44}\,2\,{}_{42}\,\mathit{offset}\,{}_{27}\,0A\,{}_{21}\cdots{}_{0}$ **A**

$tn - \mathrm{ssw}.pc + {}'\mathit{offset}$
$\{\text{where } {}'\mathit{offset} \in [-2^{14} \ldots 2^{14} - 1]\}$

(TARGET_INDEX *tn u*) $\quad {}_{64}\cdots{}_{47}\,tn\,{}_{44}\,3\,{}_{42}\,u\,{}_{37}\,000\,{}_{27}\,0A\,{}_{21}\cdots{}_{0}$ **A**

$tn - \mathrm{ssw}.pc + u$

(TARGET_RESTORE *tn u*) $\quad {}_{64}\cdots{}_{47}\,tn\,{}_{44}\,1\,{}_{42}\,u\,{}_{37}\,000\,{}_{27}\,0A\,{}_{21}\cdots{}_{0}$ **A**

$tn \leftarrow u$

(TARGET_SAVE *x tn*) $\quad {}_{64}\cdots{}_{21}\,x\,{}_{16}\,0\,{}_{13}\,0\,{}_{11}\,F\,{}_{7}\,0\,{}_{6}\,6\,{}_{3}\,tn\,{}_{0}$ **C**

$x \leftarrow \mathit{StreamStatusWord}(\mathrm{ssw}.cv, \mathrm{ssw}.tm, \mathrm{ssw}.md, tn)$

These operations establish target values for the program counter to be used by a subsequent branch operation, i.e. one of the operations from the JUMP_ family, and LEVEL_RTN.

There are eight target registers addressed by *tn*. The TARGET_DISP and TARGET_INDEX operations set the addressed target, using the values currently in the ssw.pc. TARGET_SAVE saves the StreamStatusWord with the pc replaced by the addressed target in register *x*.

Conversely, TARGET_RESTORE restores the addressed target from register *u*.

If field "priv_t0" is set in the program state descriptor, setting target zero is supervisor-privileged; This allows trap-handlers to be trustworthy. See §8.5.

When a target register is loaded, the corresponding program instructions are prefetched; see §7.2. Separating the TARGET from the JUMP allows the instruction fetch latency to be hidden.

RAISES

privileged

COUNTS AS

CNT_TARGET if not TARGET_SAVE

SEE ALSO

JUMP_, LEVEL_RTN, SSW_DISP

(**TRAP_RESTORE** *tr y*) $\quad {}_{64}\cdots{}_{21}00{}_{16}y{}_{11}tr{}_{6}22{}_{0}$ C

(trap register at *'tr*) — *y*
{**where** *'tr* ∈ [0 . . . 7]}

(**TRAP_SAVE** *x tr*) $\quad {}_{64}\cdots{}_{21}x{}_{16}tr{}_{11}03{}_{6}00{}_{0}$ C

x — (trap register at *'tr*)
{**where** *'tr* ∈ [0 . . . 7]}

These operations save and restore values to the trap registers. By convention, these operations are used by the trap handler. The TRAP_RESTORE operation does not check for poison on *y*. This exception allows the trap handler to free a register without raising a spurious poison exception.

Trap registers are allocated dynamically; see §9.2.

RAISES

(nothing)

TRAP_

(UNS_CEIL *t u*)	$_{64}\cdots_{47}\,t\,_{42}\,u\,_{37}\,18\,_{32}\,0B\,_{27}\,08\,_{21}\cdots_{0}$	A
t — unsigned integer ceiling of float *u*		
(UNS_CEIL_TEST *t u*)	$_{64}\cdots_{47}\,t\,_{42}\,u\,_{37}\,18\,_{32}\,0B\,_{27}\,09\,_{21}\cdots_{0}$	A
t — unsigned integer ceiling of float *u*		
(UNS_CHOP *t u*)	$_{64}\cdots_{47}\,t\,_{42}\,u\,_{37}\,18\,_{32}\,09\,_{27}\,08\,_{21}\cdots_{0}$	A
t — unsigned integer chop of float *u*		
(UNS_CHOP_TEST *t u*)	$_{64}\cdots_{47}\,t\,_{42}\,u\,_{37}\,18\,_{32}\,09\,_{27}\,09\,_{21}\cdots_{0}$	A
t ← unsigned integer chop of float *u*		
(UNS_FLOOR *t u*)	$_{64}\cdots_{47}\,t\,_{42}\,u\,_{37}\,18\,_{32}\,0A\,_{27}\,08\,_{21}\cdots_{0}$	A
t — unsigned integer floor of float *u*		
(UNS_FLOOR_TEST *t u*)	$_{64}\cdots_{47}\,t\,_{42}\,u\,_{37}\,18\,_{32}\,0A\,_{27}\,09\,_{21}\cdots_{0}$	A
t — unsigned integer floor of float *u*		
(UNS_NEAR *t u*)	$_{64}\cdots_{47}\,t\,_{42}\,u\,_{37}\,18\,_{32}\,08\,_{27}\,08\,_{21}\cdots_{0}$	A
t ← unsigned integer nearest float *u*		
(UNS_NEAR_TEST *t u*)	$_{64}\cdots_{47}\,t\,_{42}\,u\,_{37}\,18\,_{32}\,08\,_{27}\,09\,_{21}\cdots_{0}$	A
t ← unsigned integer nearest float *u*		
(UNS_ROUND *t u*)	$_{64}\cdots_{47}\,t\,_{42}\,u\,_{37}\,18\,_{32}\,0E\,_{27}\,08\,_{21}\cdots_{0}$	A
t ← unsigned integer round of float *u*		
(UNS_ROUND_TEST *t u*)	$_{64}\cdots_{47}\,t\,_{42}\,u\,_{37}\,18\,_{32}\,0E\,_{27}\,09\,_{21}\cdots_{0}$	A
t ← unsigned integer round of float *u*		

These operations convert floating-point numbers into unsigned integers. The roundings are directed as in IEEE Standard 754. UNS_ROUND uses the rounding mode in the ssw.

A float_invalid exception is raised when the result is negative or too large to represent. In these cases the result is reduced modulo 2^{64}.

The _TEST versions of these operations never generate carry or overflow/NaN.

RAISES

float_invalid, float_inexact

SEE ALSO

FLOAT_INT, FLOAT_UNS, INT_CEIL, INT_CHOP, INT_FLOOR, INT_NEAR, INT_ROUND, FLOAT_CEIL, FLOAT_CHOP, FLOAT_FLOOR, FLOAT_NEAR, FLOAT_ROUND

(UNS_ADD_CARRY_TEST x y z) $_{64}\cdots{}_{21}\,x\,_{16}\,y\,_{11}\,z\,_{6}\,21\,_{0}$ C

$x \leftarrow y + z + \mathrm{CV}_1.carry$, integer

(UNS_SUB_CARRY_TEST x y z) $_{64}\cdots{}_{21}\,x\,_{16}\,y\,_{11}\,z\,_{6}\,23\,_{0}$ C

$x \leftarrow y + \neg z + \mathrm{CV}_1.carry$, integer

These operations are intended to be used in multi-word integer add, subtract, and multiply. Note that carry-in is taken from CV_1, to simplify use of these operations in loops.

RAISES

(nothing)

SEE ALSO

INT_ADD, INT_SUB

UNS_ADD_CARRY_

(UNS_ADD_MUL_UPPER $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,w\,{}_{27}\,\mathrm{2C}\,{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow (u + v * w)/2^{64}$. unsigned

(UNS_ADD_MUL_UPPER_TEST $t\ u\ v\ w$) $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,w\,{}_{27}\,\mathrm{2D}\,{}_{21}\cdots{}_{0}$ **A**

$t \leftarrow (u + v * w)/2^{64}$. unsigned

The UNS_ADD_MUL_UPPER operation is used to implement multiple-precision integer multiplication.

The operation produces the high 64 bits of $u + v * w$. When u, v, and w contain 52-bit unsigned integers. the only possible effect of u on this result is via a carry from its position in the low 52 bits. The low bits of the unsigned add-multiply may be obtained from an INT_ADD_MUL operation, even though its operands are ordinarily interpreted as signed two's-complement values.

There is no UPPER_SUB_MUL because multi-word subtract-multiply does not use it.

The _TEST version of this operation never generates overflow/NaN or carry.

If v or w is outside $[-2^{53} \ldots 2^{53} - 1]$, the float_extension exception is raised.

RAISES

float_extension

SEE ALSO

INT_ADD_MUL, INT_SUB_MUL, INT_SUB_MUL_REV

(UNS_DIV t u v w) $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,w\,{}_{27}\,\mathrm{1C}\,{}_{21}\cdots{}_{0}$ A

exp — unbiased exponent of w
$temp$ — $v * w/2^{exp}$. round to floor
t — $temp * 2^{exp}$, round to floor

(UNS_DIV_TEST t u v w) $\quad {}_{64}\cdots{}_{47}\,t\,{}_{42}\,u\,{}_{37}\,v\,{}_{32}\,w\,{}_{27}\,\mathrm{1D}\,{}_{21}\cdots{}_{0}$ A

$exp \leftarrow$ unbiased exponent of w
$temp$ — $v * w/2^{exp}$, round to floor
$t \leftarrow temp * 2^{exp}$, round to floor

These operations are used to implement integer division. The product from unsigned v and SpecialFloat64 w is shifted right according to exp and rounded, producing an unsigned integer.

The _TEST version of this operation generates carry when the quotient is not exact, i.e. when the division by 2^{-exp} yields a non-zero remainder. This operation never generates overflow.

If v is outside $[0 \ldots 2^{53} - 1]$, the float_extension exception is raised and the result in t may be incorrect.

Although register u is not used in the current hardware implementation, the software requires u to contain the denominator in order to properly handle float_extension exceptions.

RAISES

float_extension

SEE ALSO

INT_DIV_CHOP, INT_DIV_FLOOR, §12.6

(UNS_LOADB *r s*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}7{}_{47}\cdots{}_{0}$ **M**

$r \leftarrow$ zero extend(byte at s), with FE_NORMAL

(UNS_LOADB_AC_DISP *r s ac disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}D{}_{47}\cdots{}_{21}ac{}_{16}disp{}_{2}3{}_{0}$ **MC**

$r \leftarrow$ zero extend(byte at $s +$ *'disp mod* 2^{48}), with *'ac*
{where *'disp* $\in [0\ldots16383]$}

(UNS_LOADB_AC_INDEX *r s ac y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}F{}_{47}\cdots{}_{21}ac{}_{16}y{}_{11}0E{}_{6}39{}_{0}$ **MC**

$r \leftarrow$ zero extend(byte at $s + y$ *mod* 2^{48}), with *'ac*

(UNS_LOADB_DISP *r s disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}D{}_{47}\cdots{}_{21}disp{}_{2}2{}_{0}$ **MC**

$r \leftarrow$ zero extend(byte at $s +$ *'disp mod* 2^{48}), with FE_NORMAL
{where *'disp* $\in [0\ldots524287]$}

(UNS_LOADB_INDEX *r s y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}F{}_{47}\cdots{}_{21}00{}_{16}y{}_{11}0E{}_{6}38{}_{0}$ **MC**

$r \leftarrow$ zero extend(byte at $s + y$ *mod* 2^{48}), with FE_NORMAL

These operations load an unsigned byte from memory. The fe_control is taken from the *ac* field if present, or forced to FE_NORMAL. If *ac* is present, its forward, data trap0, and data trap1 disable bits are used; otherwise those of *s* are used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_LOAD

SEE ALSO

STOREB, INT_LOADB

(**UNS_LOADH** *r s*) $\quad {}_{64}\cdots{}_{61}r{}_{56}s{}_{51}5{}_{47}\cdots{}_{0}$ **M**

r — zero extend(halfword at *s*), with FE_NORMAL

(**UNS_LOADH_AC_DISP** *r s ac disp*) $\quad {}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{D}{}_{47}\cdots{}_{21}ac{}_{16}sdisp{}_{4}9{}_{0}$ **MC**

r — zero extend(halfword at $s +$ *'disp mod* 2^{48}), with *'ac*
{**where** *'disp* $\in [0 \ldots 16383]$, *'sdisp* = *'disp*/4}

(**UNS_LOADH_AC_INDEX** *r s ac y*) $\quad {}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}ac{}_{16}y{}_{11}\mathrm{0A}{}_{6}39{}_{0}$ **MC**

r — zero extend(halfword at $s + 4 * y$ *mod* 2^{48}), with *'ac*

(**UNS_LOADH_DISP** *r s disp*) $\quad {}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{D}{}_{47}\cdots{}_{21}sdisp{}_{4}8{}_{0}$ **MC**

r — zero extend(halfword at $s +$ *'disp mod* 2^{48}), with FE_NORMAL
{**where** *'disp* $\in [0 \ldots 524287]$, *'sdisp* = *'disp*/4}

(**UNS_LOADH_INDEX** *r s y*) $\quad {}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}00{}_{16}y{}_{11}\mathrm{0A}{}_{6}38{}_{0}$ **MC**

r — zero extend(halfword at $s + 4 * y$ *mod* 2^{48}), with FE_NORMAL

These operations load an unsigned halfword from memory. The fe_control is taken from the *ac* field if present, or forced to FE_NORMAL. If *ac* is present, its forward, data trap0, and data trap1 disable bits are used; otherwise those of *s* are used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_LOAD

SEE ALSO

STOREH, INT_LOADH

(UNS_LOADQ *r s*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}6{}_{47}\cdots{}_{0}$ M

r — zero extend(quarterword at s), with FE_NORMAL

(UNS_LOADQ_AC_DISP *r s ac disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{D}{}_{47}\cdots{}_{21}ac{}_{16}sdisp{}_{3}5{}_{0}$ MC

r — zero extend(quarterword at s + *'disp mod* 2^{48}), with *'ac*
{**where** *'disp* $\in [0 \ldots 16383]$, *'sdisp* = *'disp*/2}

(UNS_LOADQ_AC_INDEX *r s ac y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}ac{}_{16}y{}_{11}\mathrm{0C}{}_{6}39{}_{0}$ MC

r ← zero extend(quarterword at $s + 2 * y$ *mod* 2^{48}), with *'ac*

(UNS_LOADQ_DISP *r s disp*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{D}{}_{47}\cdots{}_{21}sdisp{}_{3}4{}_{0}$ MC

r ← zero extend(quarterword at s + *'disp mod* 2^{48}), with FE_NORMAL
{**where** *'disp* $\in [0 \ldots 524287]$, *'sdisp* = *'disp*/2}

(UNS_LOADQ_INDEX *r s y*) ${}_{64}\cdots{}_{61}r{}_{56}s{}_{51}\mathrm{F}{}_{47}\cdots{}_{21}00{}_{16}y{}_{11}\mathrm{0C}{}_{6}38{}_{0}$ MC

r — zero extend(quarterword at $s + 2 * y$ *mod* 2^{48}), with FE_NORMAL

These operations load an unsigned quarterword from memory. When the destination register r is $r0$ with UNS_LOADQ, no operation is performed. The fe_control is taken from the *ac* field if present, or forced to FE_NORMAL. If *ac* is present, its forward, data trap0, and data trap1 disable bits are used; otherwise those of s are used.

RAISES

data_hw_error, data_prot, data_alignment, data_blocked

COUNTS AS

CNT_LOAD

SEE ALSO

STOREQ, INT_LOADQ

(UNS_RECIP_SHIFT x y) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}\,\mathrm{0F}_{6}\,\mathrm{00}_{0}$ C

$x \leftarrow log_2 y$, round to ceiling

(UNS_RECIP_SHIFT_TEST x y) $\quad {}_{64}\cdots{}_{21}x_{16}y_{11}\,\mathrm{0F}_{6}\,\mathrm{01}_{0}$ C

$x \leftarrow log_2 y$, round to ceiling

These operations are used to compute integer reciprocals. They compute the ceiling of the log base 2 of y. When y is zero, x is set to -1.

RAISES

(nothing)

SEE ALSO

UNS_DIV, §12.6

UNS_RECIP_SHIFT_

(UNS_SHIFT_RIGHT x y z) $_{64}\cdots{}_{21}x_{16}y_{11}z_{6}\mathrm{1C}_{0}$ C

$x - y \gg z$

(UNS_SHIFT_RIGHT_TEST x y z) $_{64}\cdots{}_{21}x_{16}y_{11}z_{6}\mathrm{1D}_{0}$ C

$x - y \gg z$

These operations shift to the right, filling in 0's on the left. Unsigned shift counts in z are taken modulo 64.

The _TEST version generates carry if a 1-bit is shifted out of w or y, and never generates overflow/NaN.

RAISES

(nothing)

SEE ALSO

INT_SHIFT_RIGHT, SHIFT_LEFT, SHIFT_PAIR_RIGHT

Chapter 12: Programming Examples

12.1 Stream Creation

[[This needs examples.]]

12.2 Forwarding Pointers

[[This needs examples.]]

12.3 Vector Loops

Consider this Fortran loop to compute the inner product:

<fortran inner product>≡

```
      do 100 i = 1, n
           sum = sum + a(i)*b(i)
100   continue
```

It could be compiled to the TERA program fragment shown below (omitting most of the preamble. and with the loop unrolled):

<tera inner product>≡

```
        (allocate-from-reg Reg 1)
        (assign-reg Reg
          ntrips sum sumoffset
          stackpointer
          ai apointer astep
          bi bpointer bstep
          )
        (equ ZERO r0)
        (inst 1 (INT_LOAD ai apointer)
                (TARGET_DISP t1 loop100)
                (REG_MOVE sum ZERO))
(label loop100)
        (inst 0 (INT_LOAD bi bpointer)
                (INT_ADD apointer apointer astep)
                (INT_ADD bpointer bpointer bstep))
        (inst 1 (INT_LOAD ai apointer)
                (FLOAT_ADD_MUL sum sum ai bi)
                (INT_ADD apointer apointer astep))
        (inst 0 (INT_LOAD bi bpointer)
                (INT_SUB_IMM_TEST ntrips ntrips 2)
                (INT_ADD bpointer bpointer bstep))
        (inst 1 (INT_LOAD ai apointer)
                (FLOAT_ADD_MUL sum sum ai bi)
                (JUMP if_ile c0 t1))
        (inst 0 (STORE_INDEX sum stackpointer sumoffset))
```

The TARGET in the first instruction sets the target's program counter. The lookahead values let pairs of instructions in the loop run in parallel. The loop achieves one flop per instruction.

Another Fortran loop of interest is the following:

<fortran inner product>≡

```
        do 200 i = 1,m
                do 100 j = 1,n
                   c(i) = c(i) + a(i,j)*b(j)
100             continue
200     continue
```

Loop transformations (strip mining and interchange) yield

<fortran inner product>≡

```
      do 201 ii = 1,m - 19,20
            lasti = ii + 19
            do 101 j = 1,n
                  do 1 i = ii,lasti
                        c(i) = c(i) + a(i,j)*b(j)
1                 continue
101         continue
201   continue
      do 202 i = lasti + 1,m
            do 102 j = 1,n
                  c(i) = c(i) + a(i,j)*b(j)
102         continue
202   continue
```

The do 1 loop may now be unrolled:

<unrolled fortran inner product>≡

```
      do 201 ii = 1,m - 19,20
            lasti = ii + 19
            do 101 j = 1,n
                  c(ii) = c(ii) + a(ii, j)*b(j)
                  c(II + 1) = c(II + 1) + a(II + 1,j)*b(j)
                  c(II + 2) = c(II + 2) + a(II + 2,j)*b(j)
                  ...
                  c(II + 19) = c(II + 19) + a(II + 19,j)*b(j)
101         continue
201   continue
      do 202 i = lasti + 1,m
            do 102 j = 1,n
                  c(i) = c(i) + a(i,j)*b(j)
102         continue
202   continue
```

The do 101 loop now corresponds to the TERA assembly language fragment below (again, omitting the preamble):

```
    <tera unrolled inner product>≡
          (allocate-from-reg Reg 1)
          (assign-reg Reg
            bj bij bpointer bstep
            aij apointer astep abigstep
            c_0 c_1 c_17 c_18 c_19
            )
          ; ...
          (inst 0 (TARGET_DISP t1 loop101))
  (label loop101)
          (inst 0 (INT_LOAD bj bpointer)
                  (INT_ADD apointer apointer abigstep)
                  (INT_ADD bpointer bpointer bstep))
          (inst 0 (INT_LOAD aij apointer)
                  (INT_ADD apointer apointer astep))
          (inst 0 (INT_LOAD aij apointer)
                  (FLOAT_ADD_MUL c_0 c_0 aij bj)
                  (INT_ADD apointer apointer astep))
          (inst 0 (INT_LOAD aij apointer)
                  (FLOAT_ADD_MUL c_1 c_1 aij bj)
                  (INT_ADD apointer apointer astep))
          ; ...
          (inst 0 (INT_LOAD aij apointer)
                  (FLOAT_ADD_MUL c_17 c_17 aij bj)
                  (INT_ADD apointer apointer astep))
          (inst 0 (INT_LOAD aij apointer)
                  (FLOAT_ADD_MUL c_18 c_18 aij bj)
                  (INT_SUB_IMM_TEST ntrips ntrips 1))
          (inst 7 (FLOAT_ADD_MUL c_19 c_19 aij bj)
                  (JUMP if_ile c0 t1))
          ; ...
```

The lookahead values of 0 throughout most of the loop could be increased by using distinct aij and apointer registers for each value of i, but this consumes more registers and thereby decreases the unrolling factor.

Minimal lookahead and an unrolling factor of 20 yields 40 flops every 22 instructions or about 1.9 flop/instruction. (This figure could be improved to about 80 flops every 42 instructions by unrolling two iterations of the j loop.) On the other hand, a sustained lookahead of 7, the maximum possible, can be had by unrolling 8 iterations of the i loop and 2 of the j loop. This approach still yields a respectable 32 flops every 18 instructions, or about 1.8 flop/instruction, with eight times fewer streams needed. Once again, the somewhat lengthy preamble is omitted; in this case, it involves loading not just 8 ci's but 8 aij's as well. The loop uses 23 registers, so it is quite reasonable to expect a compiler to unroll as fully as this:

<tera fully unrolled inner product>≡

```
    (allocate-from-reg Reg 1)
    (assign-reg Reg
      ntrips
      a0j a1j a7j apointer ajstep
      bj0 bj1 bpointer bstep c0 c1 c7)
    (define AISTEP (double 'SIZE))
    ;...
    (inst 0 (TARGET_DISP t1 loop))
(label loop)
    (inst 7 (INT_LOAD a0j apointer)
            (FLOAT_ADD_MUL c0 c0 a0j bj0)
            (INT_ADD bpointer bpointer bstep))
    (inst 7 (INT_LOAD_DISP a1j apointer AISTEP)
            (FLOAT_ADD_MUL c1 c1 a1j bj0))
    ;...
    (inst 7 (INT_LOAD_DISP a7j apointer (* 7 AISTEP))
            (FLOAT_ADD_MUL c7 c7 a7j bj0))
    (inst 7 (INT_LOAD bj0 bpointer)
            (INT_ADD apointer apointer ajstep)
            (INT_ADD bpointer bpointer bstep))
    (inst 7 (INT_LOAD a0j apointer)
            (FLOAT_ADD_MUL c0 c0 a0j bj1)
            (INT_SUB_IMM_TEST ntrips ntrips 2))
    (inst 7 (INT_LOAD_DISP a1j apointer AISTEP)
            (FLOAT_ADD_MUL c1 c1 a1j bj1))
    ;...
    (inst 7 (INT_LOAD_DISP a7j apointer (* 7 AISTEP))
            (FLOAT_ADD_MUL c7 c7 a7j bj1))
    (inst 7 (INT_LOAD bj1 bpointer)
            (INT_ADD apointer apointer ajstep)
            (JUMP if_ile c0 t1))
    ;...
```

12.4 Doubled Precision Floating-point Arithmetic

The TERA architecture provides support for 128-bit "doubled precision" floating-point arithmetic. A doubled precision representation is an ordered pair $[X, x]$ of floating-point numbers in which x is insignificant compared to X. that is. round$(X + x) = X$. The only rounding mode supported in doubled precision arithmetic is "round to nearest".

We say $[X, x]$ is the doubled precision representation of the real number ξ if X = round(ξ) and x = round(ξ − round(ξ)). It follows from the definition that round$(X + x) = X$. Testing the more significant part of a doubled precision value is sufficient because of this "normalization" property. Another important property is this: if A and B are floating-point numbers, then there is a unique, exact doubled precision representation for $A + B$.
The representation $[X, x]$ of $A + B$ is computed as follows:

<doubled precision math>≡

```
(define (doubled_add_single X x A B templ)
  (inst 0 (FLOAT_MMAX templ A B) (FLOAT_ADD X A B))
  (inst 0 (FLOAT_MMIN templ A B) (FLOAT_SUB x templ X))
  (inst 0 (FLOAT_ADD x x templ))
  )
```

A doubled precision floating-point add. $[Z, z] = [X, x] + [Y, y]$, is computed like this, where **temp1**, **temp2**, **temp3**, and **temp4** are temporary registers that are distinct from those holding Z or z:

<doubled precision math>≡

```
(define (doubled_add Z z X x Y y templ temp2 temp3 temp4)
  (let* ((A templ) (a temp2) (B temp3) (b temp4)
         (C B) (c a) (t z) (u Z)
         )
        (inst 0 (FLOAT_MMAX t X Y) (FLOAT_ADD A X Y))
        (inst 0 (FLOAT_MMIN t X Y) (FLOAT_SUB a t A))
        (inst 0 (FLOAT_ADD a a t) (FLOAT_ADD B x y))
        (inst 0 (FLOAT_MMAX t x y) (FLOAT_ADD a a B))
        (inst 0 (FLOAT_SUB b t B) (FLOAT_ADD C A a))
        (inst 0 (FLOAT_MMIN u x y) (FLOAT_SUB c A C))
        (inst 0 (FLOAT_ADD c c a) (FLOAT_ADD b b u))
        (inst 0 (FLOAT_ADD c c b))
        (inst 0 (FLOAT_ADD Z C c))
        (inst 0 (FLOAT_SUB z C Z))
        (inst 0 (FLOAT_ADD z z c))
        )
  )
```

Doubled precision floating-point subtract is similar, with the *y* and *Y* operands explicitly negated.

<doubled precision math>≡

```
(define (doubled_sub Z z X x Y y temp1 temp2 temp3 temp4)
  (let* ((A temp1) (a temp2) (B temp3) (b temp4)
         (C B) (c a) (t z) (u Z)
         )
        (inst 0 (BIT_MASK a 63 63) (FLOAT_SUB A X Y))
        (inst 0 (BIT_XOR a Y a) (BIT_XOR u y a))
        (inst 0 (FLOAT_MMAX t X a) (FLOAT_SUB B x y))
        (inst 0 (FLOAT_MMIN t X a) (FLOAT_SUB a t A))
        (inst 0 (FLOAT_MMAX t x u) (FLOAT_ADD a a t))
        (inst 0 (FLOAT_SUB b t B) (FLOAT_ADD a a B))
        (inst 0 (FLOAT_ADD C A a))
        (inst 0 (FLOAT_MMIN u x u) (FLOAT_SUB c A C))
        (inst 0 (FLOAT_ADD c c a) (FLOAT_ADD b b u))
        (inst 0 (FLOAT_ADD c c b))
        (inst 0 (FLOAT_ADD Z C c))
        (inst 0 (FLOAT_SUB z C Z))
        (inst 0 (FLOAT_ADD z z c))
        )
  )
```

The doubled precision product $[Z, z] = [X, x] \times [Y, y]$ is computed as follows, where temp1 and temp2 must be temporary registers distinct from those holding Z, z, X, x, Y, or y. This code relies on the fact that the floating-point multiply-add operations only round once.

<doubled precision math>≡

```
(define (doubled_mul Z z X x Y y temp1 temp2)
  (let* ((a temp1) (b temp2)
         (c temp1) (d temp2) (e z) (f temp2)
         (ZERO r0)
         )
        (inst 0 (FLOAT_MUL_LOWER a ZERO X y))
        (inst 0 (FLOAT_MUL_LOWER b ZERO x Y))
        (inst 0 (FLOAT_ADD_MUL c ZERO X Y)
                (FLOAT_ADD d a b))
        (inst 0 (FLOAT_MUL_LOWER e c X Y))
        (inst 0 (FLOAT_ADD f d e))
        (inst 0 (FLOAT_MMAX z c f) (FLOAT_ADD Z c f))
        (inst 0 (FLOAT_MMIN c c f) (FLOAT_SUB z z Z))
        (inst 0 (FLOAT_ADD z z c))
        )
  )
```

An important special case occurs when products of working precision numbers X and Y are to be computed in doubled precision $[Z, z]$:

<doubled precision math>≡

```
(define (doubled_single_mul Z z X Y)
  (let ((ZERO r0))
       (inst 0 (FLOAT_ADD_MUL Z ZERO X Y))
       (inst 0 (FLOAT_MUL_LOWER z Z X Y))
       )
  )
```

12.5 Floating-point Division and Square Root

While directly implementing floating-point division was deemed infeasible, the TERA architecture provides support for correctly rounded IEEE division. Starting with a reciprocal approximation, the adder-multiplier is used to compute a correctly rounded quotient.
The floating-point divide $q = x/y$ is computed like this, where q, x and y are held in registers and `temp1` is an additional temporary register which must be distinct from the first three.

<floating divide>≡

```
(define (float_divide q x y temp1 temp2)
  (let ((r temp1) (e temp2))
       (float_reciprocal r y q)
       (inst 0 (FLOAT_DIV_APPROX q y x r))
       (inst 0 (FLOAT_DIV_ERROR e x y q))
       (inst 0 (FLOAT_DIV q q e r))
       )
  )
```

<floating divide>≡

```
(define (float_reciprocal r y temp1)
  (let ((e temp1))
       (inst 0 (FLOAT_RECIP_APPROX r y)) ; traps if y is denorm
       (inst 0 (FLOAT_RECIP_ERROR e y r))
       (inst 0 (FLOAT_ITER r r e r))
       (inst 0 (FLOAT_RECIP_ERROR e y r))
       (inst 0 (FLOAT_ITER r r e r))
       )
  )
(define (float_reciprocal_denorm r y)
  (let ((temp1 r))
       (inst 0 (INT_IMM temp1 1074))
       (inst 0 (FLOAT_SCALB temp1 y temp1))
       (inst 0 (INT_RECIP_APPROX r temp1))
       )
  )
```

The first FLOAT_RECIP_APPROX gives a reciprocal that is accurate to 14 bits. The FLOAT_-RECIP_ERROR and FLOAT_ITER pairs form a Newton iteration, which raises the accuracy to 28 and then 54 bits. The product of the reciprocal r and the numerator x is a quotient q correct to 1 ulp. The final instructions compute a remainder to correctly round q and deliver the final quotient. When the divisor y is denormalized, the trap handler should execute the float_reciprocal_denorm code sequence to compute the correct initial reciprocal approximation.

Computing the square root is performed similarly. The square root $q = \sqrt{y}$ is computed like this, where q and y are held in registers, and templ is an additional temporary register which must be distinct from the first two.

```
    <floating square root>≡
(define (float_sqrt q y templ temp2)
  (let ((r templ) (e temp2)
        (_float_sqrt q y r e)
        (inst 0 (FLOAT_SQRT_ERROR_TEST e y q q))
        (inst 0 (FLOAT_SQRT q q e r))
        )
  )
)
(define (_float_sqrt q y r e)
       (inst 0 (FLOAT_RSQRT_APPROX r y)) ; traps if y is denorm
       (inst 0 (FLOAT_SQRT_APPROX_TEST q y y r))
       (inst 0 (FLOAT_RSQRT_ERROR_TEST e y q r))
       (inst 0 (FLOAT_ITER r r e r))
       (inst 0 (FLOAT_SQRT_APPROX_TEST q y y r))
       (inst 0 (FLOAT_RSQRT_ERROR_TEST e y q r))
       (inst 0 (FLOAT_ITER r r e r))
       (inst 0 (FLOAT_SQRT_APPROX_TEST q y y r))
  )
(define (float_rsqrt_denorm r y)
  (let ((templ r))
       (inst 0 (INT_IMM templ 1022))
       (inst 0 (FLOAT_SCALB templ y templ))
       (inst 0 (INT_RSQRT_APPROX r templ))
       )
  )
```

Here, the initial approximation to the reciprocal square root is correct to at least 14 bits. The FLOAT_SQRT_APPROX_TEST uses the reciprocal root to compute an estimate of the square root. The two iterations improve the accuracy to the necessary 54 bits. The final operation computes the correct rounding for the delivered result. When the argument y is denormalized, the trap handler should execute the float_rsqrt_denorm code sequence to compute the correct initial reciprocal square root approximation.

12.6 Integer Division

The TERA architecture provides support for 64-bit integer division, including both signed and unsigned integer data types. The current implementation supports 53-bit integer division in hardware and longer operands in software. For signed division, we provide instructions to allow the quotient to be rounded toward zero by chopping, as in FORTRAN, or rounded toward negative infinity, so that $q = \lfloor \frac{x}{y} \rfloor$.
The FORTRAN integer divide $q = x/y$ is computed like this, where q, x and y are held in registers, and `temp1` is an additional temporary register, which must be distinct from the first three.

<integer divide>≡

```
(define (int_reciprocal r y temp1 temp2)
  (let ((fy temp2)
        (e temp1)
       )
       (inst 0 (FLOAT_INT fy y))
       (inst 0 (INT_RECIP_APPROX r fy))
       (inst 0 (INT_RECIP_ERROR e fy r))
       (inst 0 (FLOAT_ITER r r e r))
       (inst 0 (INT_RECIP_ERROR e fy r))
       (inst 0 (FLOAT_ITER r r e r))
       (inst 0 (INT_DIV_CHOP e y y r)
             (INT_ADD_IMM r r 1))
       (inst 0 (INT_SUB r r e))
       )
  )
```

The first reciprocal approximation is correct to 14 bits. Two iterations raise that accuracy to 54 bits. The fix step is needed to guarantee that *r* is correctly rounded to the ceiling of the reciprocal. Finally, the divide instruction multiplies by the reciprocal and shifts right to compute the correct quotient.

<integer divide>≡

```
(define (int_divide_chop q x y temp1 temp2)
  (let ((r temp1)
       )
      (int_reciprocal r y q temp2)
      (inst 0 (INT_DIV_CHOP q y x r))
      )
  )
(define (int_divide_chop_test q x y temp1 temp2)
  (let ((r temp1)
       )
      (int_reciprocal r y q temp2)
      (inst 0 (INT_DIV_CHOP_TEST q y x r))
      )
  )
```

Signed integer division with floored rounding is analogous, except the final INT_DIV_CHOP is replaced with INT_DIV_FLOOR so that the sign of the (implied) remainder agrees with the denominator rather than the numerator.

<integer divide>≡

```
(define (int_divide_floor q x y temp1 temp2)
  (let ((r temp1)
       )
      (int_reciprocal r y q temp2)
      (inst 0 (INT_DIV_FLOOR q y x r))
      )
  )
(define (int_divide_floor_test q x y temp1 temp2)
  (let ((r temp1)
       )
      (int_reciprocal r y q temp2)
      (inst 0 (INT_DIV_FLOOR_TEST q y x r))
      )
  )
```

The unsigned quotient is computed similarly:

```
  <unsigned divide>≡
(define (uns_reciprocal r y temp1 temp2)
  (let ((fy   temp2)
        (e    temp1)
        )
       (inst 0 (FLOAT_UNS fy y))
       (inst 0 (INT_RECIP_APPROX r fy))
       (inst 0 (INT_RECIP_ERROR e fy r))
       (inst 0 (FLOAT_ITER r r e r))
       (inst 0 (INT_RECIP_ERROR e fy r))
       (inst 0 (FLOAT_ITER r r e r))
       (inst 0 (UNS_DIV e y y r)
               (INT_ADD_IMM r r 1))
       (inst 0 (INT_SUB r r e))
       )
  )
```

```
  <unsigned divide>≡
(define (uns_divide q x y temp1)
  (let ((r temp1)
        )
       (uns_reciprocal r y q)
       (inst 0 (UNS_DIV q y x r))
       )
  )
(define (uns_divide_test q x y temp1)
  (let ((r temp1)
        )
       (uns_reciprocal r y q)
       (inst 0 (UNS_DIV_TEST q y x r))
       )
  )
```

Note that when the divisor is constant, the sequence simplifies down to one compute instruction (and one constant which must be materialized). For example, the following two instructions compute $q = x/20$.

<integer divide by 20>≡

```
(define (int_divide_20 q x)
  (let* ((r q))
        (const FIFTH (+ (- (quotient (+ (expt 2 56) 4) 5) (expt 2 53))
                (* (- 510 3) (expt 2 53))))
        (inst 0 (LOAD_DISP r LINKAGE_PTR FIFTH))
        (inst 0 (INT_DIV_CHOP q r0 x r))
        )
  )
```

Chapter 13: I/O Processor Introduction

The I/O processor contains four instruction streams and a control word. The control word is used to assign a segment descriptor for fetching instructions and initializing the streams. The four streams have dedicated purposes: memory load, memory store, HIPPI input and HIPPI output. For each stream, instructions are fetched and executed in a linear fashion. If an exceptional event occurs during the execution of an instruction, the stream performs a link operation with the driver to inform it of the exception. A link operation is defined as a stream status word production, followed by a program counter consumption. The driver program must use consumer/producer semantics on the link address and the new control word address, respectively. The I/O processor ignores the forward, data trap, and memory full bits when fetching instructions.

- The *memory load* stream loads from data or I/O memory into the outbound data buffer.
- The *memory store* stream stores from the inbound data buffer into data or I/O memory.
- The *HIPPI output* stream transfers bursts of data from the outbound data buffer out through the HIPPI interface.
- The *HIPPI input* stream transfers bursts of received data into the inbound data buffer.

The HIPPI interface section of the IOP is designed to conform with the physical layer specification of the ANSI X3T9.3 committee. The basic HIPPI clock rate is 25 Mhz, and the timeout clock period is 1 microsecond. The longest timeout period is 16.8 seconds. The HIPPI out section of the IOP has an external. fixed 50 Mhz clock which is used to generate the 25 Mhz timing and the IOP timeout clock. The HIPPI in section synchronizes to the source clock for its connection, yet uses the fixed timeout clock generated by the HIPPI out section.

Each half of the IOP can operate as either a 32- or 64-bit HIPPI channel. The channel width can be selected when making each connection. The load and store streams operate only on 64-bit words.

On power up, the IOP must not request or accept any connections until it has been initialized. To satisfy this requirement, some (external) power on reset circuit is needed. In addition, connections must be inhibited during scan.

13.1 Link Status Word

Each stream in the I/O processor produces a link status word when it links. That word generally indicates the reason for linking, and the program counter at which the link occurred. The program counter in both link words is a byte offset into the instruction segment. The lower three bits are ignored. since the IOP only performs full word. aligned. instruction fetches. The field "exception" is set whenever one of the exceptions in bits 53 to 39 is set. The field "status_link" is set whenever the field "pc" is not a program counter, but instead is some data such as an Ifield or error offset.

IOP Status Word

Valid	Bits	Wd	Field Name	Type	Description
IOPStatusWord: Exceptions					
LSOI	63	1	exception	Flag	Exception
LSI	62	1	status_link	Flag	Status result
	61–56	6	os_field	Uns	Reserved for O/S use; IOP writes 0
	55–54	2	0		*reserved*
LSOI	53	1	forced_link	Flag	Forced link Operation
LSOI	52	1	link_error	Flag	Link Error
LSOI	51	1	p_limit_error	Flag	Instruction Segment Limit Error
LSOI	50	1	p_-unimplemented_-address	Flag	Instruction Address unimplemented on resource
LSOI	49	1	p_uncorrectable_-error	Flag	Uncorrectable Instruction Error
LSOI	48	1	illegal_op_code	Flag	Illegal OP code
S	47	1	packet_end	Flag	Packet End exception
LSI	46	1	limit_error	Flag	Segment Limit Error
LSI	45	1	unimplemented_-address	Flag	Address unimplemented on resource
LSOI	44	1	uncorrectable_-error	Flag	Uncorrectable Data Error
OI	43	1	connect_lost	Flag	Connect lost
OI	42	1	interconnect_lost	Flag	Interconnection lost
OI	41	1	bad_connect	Flag	Unable to request connection
OI	40	1	timeout	Flag	Time out
O	39	1	no_connection	Flag	No connect before OUT_PACKET
	38–35	4	0		*reserved*
IOPStatusWord: Status					
LSOI	34–33	2	Stream_identity	IopStream	Stream Identity
LSOI	32	1	loopback	Flag	Loopback
IOPStatusWord: PC					
LSOI	31–0	32	pc	Uns	Program Counter or Ifield

The field "Stream_identity" is encoded using the following enumeration.

Name	Value	Meaning
IopStream		
IOP_LOAD	0	Load stream
IOP_OUT	1	Out stream
IOP_STORE	2	Store stream
IOP_IN	3	In stream

The link status words are arranged at word indices 0 through 7 in the IOP instruction segment.

13.1 Link Status Word IopStream

Each stream has a pair. status and next pc. These are laid out in the following order: load_status, load_next_pc, out_status. out_next_pc, store_status, store_next_pc, in_status, in_next_pc.

Chapter 14: I/O Operation Descriptions

I/O Processor programs have the same syntactic form as Lisp expressions. The CPU instructions are composed of several operations, IOP instructions always contain exactly one operation. Therefore, the INST wrapper is not needed in IOP assembly programs.

(INST_SEGMENT *dist_en m_type unit limit base*)

$_{64}00_{56}\ 0_{55}\ dist\_en_{54}\ m\_type_{52}\ _{48}0\ unit_{40}\ _{39}0\ limit_{24}\ _{19}0\ base_{0}$ I

InstSegment — immediate data

The segment word holds the data address translation descriptor for IOP instructions. The format is the same as data map entries in the processor except that the load and store levels and locked bit are omitted.

The I/O Processor is reset or initialized through the segment word. The segment word is located at word offset 0 for the logical unit number assigned to the IOP. Whenever the segment word is written to, all four streams perform a link operation in the new segment. As there is only one program segment for all four streams, the IOP should be in an idle or suspect state before changing the segment descriptor. A write of the segment word also resets the inbound and outbound data buffers. Table 1 shows the bit allocation for the stream status returned by a link operation.

The p_limit_error. p_unimplemented_address, and p_uncorrectable_error exceptions during a link operation cause the stream to retry the link operation, possibly indefinitely.

Instructions are provided so that one stream may cause another stream to link. Generally, streams forced to link will link between instructions. However, a stream will abort an indefinite wait to link, so that deadlocked streams may be interrupted and reset.

IOP_RESET

(LOAD_LINK) $_{64}$10$_{56}$00000000000000$_{0}$ I

Link Load stream

(LOAD_LINK_OUT) $_{54}$14$_{56}$00000000000000$_{0}$ I

Link Out stream

The LINK operation forces the memory load stream to perform a link operation with the device driver. The link operation pair for the memory load stream is located at word offset 0 and 1 of the program segment for IOP code. No exception is raised.

The LOAD_LINK_OUT operation forces the out stream to perform a link operation. The out stream will link at the next opportunity, generally after completing the current instruction, but potentially by aborting an instruction in progress. The forced link bit will be set in the out stream's status word.

RAISES

(nothing)

(**LOAD_SEGMENT** *dist_en m_type unit limit base*)

$_{64}11_{56}\,0_{55}\,\mathit{dist\_en}_{54}\,\mathit{m\_type}_{52}\,0_{48}\,\mathit{unit}_{40}\,0_{39}\,\mathit{limit}_{24}\,0_{19}\,\mathit{base}_{0}$ I

LoadSegment — immediate data

This operation loads an address translation descriptor to be used by the memory load stream for fetching data. The format is the same as data map entries in the processor except that the load and store levels and locked bit are omitted.

There are no exceptions for the LOAD_SEGMENT opcode.

RAISES

(nothing)

LOAD_SEGMENT_

(LOAD_ERR_OFFSET) $_{64}12_{56}000000000000000_{0}$ I

Status.pc — Error offset

This operation always links. If there are no masking exceptions, this operation stores the offset of the load request which first resulted in an uncorrectable_error, limit_error, or unimplemented_-address exception in the pc field of the link status word. The status_link flag is set. If multiple errors are encountered, only the first is reported. Note that the offset returned may not be the lowest offset which resulted in an exception. If no errors were present, LOAD_ERR_OFFSET returns the next offset that load stream will issue to the network. This operation is used for diagnosing the failure of a load instruction.

RAISES

status_link

(LOAD_FLUSH) $_{64}13_{56}00000000000000_{0}$ I

Flush outbound data buffer

The flush operation clears all data in the outbound data buffer. This operation should be used only when the HIPPI out stream is not connected. If the flush operation is used while the HIPPI out stream is connected, the connection will be dropped.

RAISES

(nothing)

LOAD_FLUSH_

(LOAD_END_PACKET) $_{64}15_{56}00000000000000_{0}$ I

Mark end of indeterminate length packet

The end_packet operation indicates to the out stream that the contents of the buffer are the last words of an indeterminate length packet. This command should only be used with an OUT_-PACKET command of length 0. The load stream will wait until the outbound buffer is emptied or the OUT_PACKET command otherwise terminates before continuing with the execution of the next instruction.

RAISES

(nothing)

(**LOAD_DATA** *start_offset end_offset*) $_{64}2_{60}$ *start_offset*$_{32}$ *end_offset*$_{0}$ I

Load data

(**LOAD_IMAGE** *start_offset end_offset*) $_{64}3_{60}$ *start_offset*$_{32}$ *end_offset*$_{0}$ I

Load image

Load the 64-bit data and state beginning at the start_offset and continuing through all words up to end_offset minus one, inclusive. The low-order three bits of the start_offset and end_offset are ignored. Thus, byte addresses may be used without need for shifting.

The load stream interprets the end_offset modulo 2^{28}. To include the last word in a segment, a 10000000_{16} or 0 may be used as the end_offset.

All forwarding, data traps. and full/empty operations are disabled during the load operation. The load stream issues LOAD_STATE request to the memory resources, which respond with both the data and the access state stored at the given address.

If _IMAGE is used, the outgoing data buffer packs the control access fields for sixteen consecutive data words into a 64-bit state word and inserts the state word into the data stream after its respective data words. When _IMAGE is used, the number of words to load from memory must be a multiple of 16. In addition, the number of non-packed data words loaded from memory before the _IMAGE operation occurs must be a multiple of 16. Note that these operations may be used together to build a packet with a DATA header and IMAGE payload.

This instruction need not abort due to a forced link unless there are no free buffers into which to load.

This instruction will wait indefinitely for free space in the outbound buffer. Exceptions for the LOAD_DATA/IMAGE operations are forced_link, uncorrectable_error, limit_error, or unimplemented_address.

RAISES

forced_link. uncorrectable_error, limit_error, unimplemented_address

LOAD_

(STORE_LINK) $_{64}80_{56}00000000000000_{0}$ I

Link Store stream

(STORE_LINK_IN) $_{64}84_{56}00000000000000_{0}$ I

Link In stream

The STORE_LINK operation forces the memory store stream to perform a link operation with the device driver. The link operation pair for the memory store stream is located at word offset 4 and 5 of the program segment for IOP code. No exception is raised.

The STORE_LINK_IN operation forces the in stream to perform a link operation. The current instruction of the in stream may be interrupted. The forced link bit will be set in the in stream's status word.

RAISES

(nothing)

(**STORE_SEGMENT** *dist_en m_type unit limit base*)

${}_{64}81_{56}\,0_{55}\,\mathit{dist\_en}_{54}\,\mathit{m\_type}_{52}\,0_{48}\,\mathit{unit}_{40}\,0_{39}\,\mathit{limit}_{24}\,0_{19}\,\mathit{base}_{0}$ I

StoreSegment — immediate data

This operation loads an address translation descriptor to be used by the memory store stream for writing data. The format is the same as data map entries in the processor except that the load and store levels and locked bit are omitted.

RAISES

(nothing)

(STORE_REPLICATE *start_offset end_offset*) $_{64}9_{60}$*start_offset*$_{32}$*end_offset*$_{0}$ I

Store fill data

This operation is only valid while the IOP is in loopback mode. The LOAD stream should fetch the word that is to be replicated. The STORE stream duplicates the item and its access state to all the locations between start_offset and end_offset minus one. inclusive. The low-order three bits of the start_offset and end_offset are ignored. Thus, byte addresses may be used without need for shifting. If multiple items are fetched by the LOAD stream, only the first item is removed from the outbound buffer.

The store stream interprets the end_offset modulo 2^{28}. To include the last word in a segment, a 10000000_{16} or 0 may be used as the end_offset.

This instruction need not abort due to a forced link unless the inbound buffer is completely empty.

This instruction will wait indefinitely for data to appear in the inbound buffer. Exceptions for the STORE_REPLICATE operations are uncorrectable_error, limit_error or unimplemented_address.

RAISES

uncorrectable_error, limit_error, unimplemented_address

(STORE_ERR_OFFSET) $_{64}83_{56}00000000000000_{0}$ I

Status.pc — Error offset

This operation always links. If there are no masking exceptions, this operation stores the offset of the store request which first resulted in an uncorrectable_error, unimplemented_address, packet_-end, or limit_error exception in the pc field of the link status word. The status_link flag is set. If multiple errors are encountered, only the first is reported. Note that the offset returned may not be the lowest offset which resulted in an exception. If no errors were present, STORE_ERR_OFFSET returns the next offset that the store stream would have issued to the network. This operation is used for diagnosing the failure of a store instruction.

RAISES

status_link

(STORE_FLUSH) $_{64}82_{56}00000000000000_{0}$ **I**

Flush inbound data buffer

The flush operation clears all data in the inbound data buffer. This operation should only be used when the HIPPI input stream is not receiving a packet. If the flush operation is used while the HIPPI input stream is receiving data, parts of the incoming data packet may be lost.

RAISES

(nothing)

(STORE_END_PACKET) $_{64}85_{56}00000000000000_{0}$ I

Handle packet end clean up

This operation flushes all data for the current packet that has not already been stored, finishing when end of packet is received. Thus, fill data at the end of a packet can be disposed of using this operation. At the extreme, a whole packet of data will be flushed if no STORE_DATA or STORE_IMAGE operations are placed between successive STORE_END_PACKET operations.

In addition, the uncorrectable_error exception is raised if an uncorrectable_error was detected during the reception of this packet by the in stream. Note that a STORE_END_PACKET must be issued for each packet received by the HIPPI IN stream. This operation always links on completion.

RAISES

uncorrectable_error

STORE_END_PACKET_

(STORE_END_SEGMENT) $_{64}86_{55}00000000000000_{0}$ I

Validate partial packet data

This operation ensures that the uncorrectable_error exception is raised if an uncorrectable_error was detected during the reception of any of the data for this packet which has already been stored. Thus. if the end of a burst has not yet been received, but the initial data in the burst has been stored. this operation will wait until the LLRC check at the end of the burst has been performed. This operation always links on completion.

RAISES

uncorrectable_error

(STORE_DATA *start_offset end_offset*) $_{64}$A$_{60}$*start_offset*$_{32}$*end_offset*$_{0}$ I

Store data

(STORE_IMAGE *start_offset end_offset*) $_{64}$B$_{60}$*start_offset*$_{32}$*end_offset*$_{0}$ I

Store image

Store the 64-bit data and state beginning at the start_offset and continuing through all words up to end_offset minus one, inclusive. If an end of packet is signaled before all the indicated words are received and stored. the packet_end exception will be raised. The low-order three bits of the start_offset and end_offset are ignored. Thus, byte addresses may be used without need for shifting.

The store stream interprets the end_offset modulo 2^{28}. To include the last word in a segment, a 10000000_{16} or 0 may be used as the end_offset.

When _DATA is used, the access control states will be set to full, no forwarding, no traps. When _IMAGE is used, the access control states are unpacked from the inbound buffer. When STORE_-IMAGE is used, the number of words to store to memory must be a multiple of 16. The store stream removes every 17th word and uses the data contained in it to generate the access control states for the preceding 16 words. An uncorrectable_error may be caused by failure of the inbound buffer or bad incoming HIPPI data. Note that these operations may be used together to scatter store a packet with a DATA header and IMAGE payload.

These instructions will wait indefinitely for data to appear in the inbound buffer. Exceptions for the STORE_DATA/IMAGE opcodes are uncorrectable_error, unimplemented_address, packet_end, or limit_error. The unimplemented address and limit_error exceptions should only occur due to program errors.

RAISES

uncorrectable_error. unimplemented_address, packet_end, limit_error

STORE_

(OUT_LINK) $_{64}41_{56}00000000000000_{0}$ I

Link Out stream

(OUT_LINK_LOAD) $_{64}48_{56}00000000000000_{0}$ I

Link Load stream

The OUT_LINK operation forces the HIPPI output stream to perform a link operation with the device driver. The link operation pair for the HIPPI output stream is located at word offset 2 and 3 of the program segment for IOP code. No exception is raised.

The OUT_LINK_LOAD operation forces the load stream to perform a link operation. The current instruction of the load stream may be interrupted. The forced link bit will be set in the load stream's status word.

(OUT_RING *swap width timeout Ifield*) $\quad {}_{64}2_{61}\,\mathit{swap}_{60}\,1_{57}\,\mathit{width}_{56}\,\mathit{timeout}_{32}\,\mathit{Ifield}_{0}$ I

Request a connection on the HIPPI interface

The HIPPI request signal is asserted and the 32-bit I-field is placed on the HIPPI data path bits 31 to 0. All zeros are placed on the remaining HIPPI data bits 63 to 32. The connect signal must initially be deasserted before request can be asserted. If the connect signal was asserted when the OUT_RING operation is executed a bad_connect exception is raised.

This instruction waits until the connect signal is asserted by the destination or timeout occurs. Once the connect signal is asserted, the IOP stops transmission of the I-field. The stream then tries to detect a rejected connection. If during this time a ready pulse is received, the stream assumes the connection was accepted. If the connect line is deasserted before a ready pulse is received, the connection was rejected and a connect_lost exception is generated.

The width field selects either a 32- or 64-bit channel width. When width is asserted, the IOP operates in 64-bit HIPPI mode. When width is deasserted, the order of the 32-bit words may be reversed by setting the swap bit.

Exceptions for the OUT_RING opcode are timeout, connect_lost, interconnect_lost or bad_connect.

RAISES

timeout, connect_lost, interconnect_lost, bad_connect

OUT_RING_

(OUT_LOOPBACK *timeout)* $_{64}$4A$_{56}$ *timeout*$_{32}$00000000$_{0}$ I

Request loopback connection

The HIPPI-output stream indicates to the HIPPI-input stream that it wishes to be in loopback mode. If the HIPPI-input stream acknowledges with a similar IN_LOOPBACK command before timeout occurs. OUT_LOOPBACK mode is established. When the IOP is in loopback mode, any data and access control state present in the outbound buffer is available for transfer into the inbound buffer. Loopback mode continues until terminated by the HIPPI-input stream or via LOAD_LINK_OUT.

The exceptions for the OUT_LOOPBACK opcode are timeout, connect_lost, and force_link.

RAISES

timeout, connect_lost, force_link

(OUT_LOOPMODE) $_{64}$4A$_{56}$000001$_{32}$00000000$_{2}$1$_{0}$ I

Select local serial loopback

The HIPPI-output stream indicates serial link logic that it wishes to be in local serial loopback mode. This mode remains in effect until cleared by a subsequent OUT_LOOPBACK instruction.

RAISES

(nothing)

OUT_LOOPMODE_

(**OUT_DISCONNECT** *timeout*) ${}_{64}44_{56}\,\mathit{timeout}_{32}00000000_{0}$ I

Break connection

This operation completes the HIPPI connection by deasserting the request signal and waiting for the destination to acknowledge by deasserting the connect signal.

If a disconnect is issued while the IOP is in loopback mode, both the HIPPI input and output streams will exit loopback mode.

The exception for the OUT_DISCONNECT opcode is timeout.

(OUT_CANCEL *timeout*) $_{64}45_{56}$ *timeout*$_{32}$00000000$_{0}$ I

Wait for late connect response

Wait for the connect signal to be asserted or for timeout to occur. After a RING fails, this instruction may be used to wait up to a round trip delay for a late connect response to the previous request assertion.

The exception for OUT_CANCEL is timeout.

RAISES

timeout

OUT_CANCEL_

(OUT_INTERCONNECT *width timeout*) $_{64}$27$_{57}$ *width*$_{56}$ *timeout*$_{32}$00000000$_{0}$ I

Wait for interconnect asserted

Wait for the interconnection lines for the specified HIPPI width to settle to an active state or for timeout to occur.

The exception for the OUT_INTERCONNECT opcode is timeout.

RAISES

timeout

(OUT_DELAY *timeout*) $_{64}47_{56}$ *timeout* $_{32}00000000_{0}$ I

Wait

Wait for timeout to occur; then fetch the next instruction.

There are no exceptions for OUT_DELAY.

RAISES

(nothing)

OUT_DELAY_

(OUT_RESET *timeout*) $_{64}47_{56}\textit{timeout}_{32}00000001_{0}$ I

Send RESET to serial link

Assert RESET to the serial link for the timeout period. then fetch the next instruction.

There are no exceptions for OUT_RESET.

RAISES

(nothing)

(OUT_PACKET *timeout size*) $_{64}49_{56}$ *timeout*$_{32}$ *size*$_{0}$ I

Send packet

This operation sends a packet containing a number of 256 word bursts. and one final short burst, if needed. out on the HIPPI interface.

The number of eight-bit bytes to send is specified by the *size* field. which must be a multiple of eight (*size* is 8 * *words*). If LOAD_IMAGE is used, the number of words to fetch from memory must be a multiple of 16. and *size* is calculated by (17 * *words*)/2.

The maximum packet length is $2^{32} - 8$ bytes. A zero length packet will cause the out stream to transmit an indeterminate length packet. A indeterminate length packet is completed by the LOAD stream issuing a LOAD_END_PACKET instruction. When the out stream receives the end packet indication from the load stream it empties the outbound buffer and then deasserts the packet signal. The instruction then completes and the next operation is fetched. Note that this handling only allows one indeterminate length packet to be present in the outbound buffer at a time.

Output flow control is handled automatically by the IOP. The memory load stream produces data into the outbound data buffer and the output stream sends the data out as allowed by the ready signals from the destination. The out stream will wait indefinitely for data to appear in the outbound buffer. Errors in the buffer data will raise the uncorrectable_error exception. When such bad data is encountered. the LLRC of the burst will be corrupted to guarantee that the receiver discards the packet.

During the OUT_PACKET operation, a LOAD_LINK_OUT operation will only allow any bursts already in the outbound buffer to be sent. If there are no more bursts present and the OUT_-PACKET operation has not completed, the packet will be truncated with a forced link exception. Note that this will cause a packet shorter than the encoded size to be transmitted.

If the connection is lost. the operation is aborted and a connect_lost exception is raised. If the connection was not present before packet is asserted. then the no_connection exception will be raised instead of the connect_lost exception. The timeout field specifies the maximum time to wait for a ready pulse when a burst is ready to send, but the ready counter is zero.

Exceptions for the OUT_PACKET opcode are timeout. uncorrectable_error, no_connection. or connect_lost.

RAISES

timeout. uncorrectable_error, no_connection. connect_lost

OUT_PACKET_

(IN_LINK) $_{64}$C1$_{56}$00000000000000$_{0}$ I

Link In stream

(IN_LINK_STORE) $_{64}$CB$_{56}$00000000000000$_{0}$ I

Link Store stream

The IN_LINK operation forces the HIPPI input stream to perform a link operation with the device driver. The link operation pair for the HIPPI input stream is located at word offset 6 and 7 of the program segment for IOP code. No exception is raised.

The IN_LINK_STORE operation forces the store stream to perform a link operation. The current instruction of the store stream may be interrupted. If the store stream is executing a STORE_DATA, STORE_IMAGE, or STORE_REPLICATE operation, it will delay the link operation until the in buffer is empty. The forced link bit will be set in the store stream's status word.

(IN_LISTEN *timeout*) $_{64}$C5$_{56}$*timeout*$_{32}$00000000$_{0}$ I

Wait for connection request

This operation waits until a connection is requested or a timeout occurs. The hardware continuously monitors the REQUEST signal, and registers a connection request when the signal transitions from deasserted to asserted. If a timeout occurs, the bad_connect exception is raised if REQUEST is asserted, but has not transitioned to deasserted since the last disconnection. When REQUEST is deasserted, timeout simply raises the timeout exception.

If no exception occurs, the current value on the HIPPI input data pins is stored in the pc field of the link status word and the status_link flag is set. The IN stream then waits for a new pc, completing a link operation.

While an IN_LISTEN is waiting for a connection request, it will immediately abort with a forced link exception upon executing STORE_LINK_IN in the STORE stream.

The exceptions for the IN_LISTEN opcode are timeout, bad_connect, or interconnect_lost.

RAISES

timeout, bad_connect, interconnec_lost, status_link

IN_LISTEN_

(IN_REJECT) $_{64}$C4$_{56}$00000000000000$_{0}$ I

Reject connection request

This instruction asserts the connect signal. then deasserts it after eight HIPPI clock cycles. If the request signal is already deasserted. connect is not asserted and the connect_lost exception is raised.

The exception to the IN_REJECT opcodes is connect_lost.

RAISES

connect_lost

(IN_ACCEPT *swap width timeout*) ${}_{64}6_{61}\,\mathit{swap}_{60}\,4_{57}\,\mathit{width}_{56}\,\mathit{timeout}_{32}\,00000000_{0}$ I

Accept connection request

This instruction asserts the connect signal. If the request signal is deasserted, connect is not asserted and the connect_lost exception is raised.

The width field indicates the channel width of the HIPPI in stream. When width is asserted, a 64-bit HIPPI channel is used. If no bursts are received within a timeout period, the timeout exception is raised. When width is deasserted, the order of the 32-bit words may be reversed by setting the swap bit.

Input flow control is handled automatically by the IOP. A ready indication will only be signaled to the source when the IOP can guarantee buffer space (in the IOP or in memory) to hold the enabled burst. The in stream will wait indefinitely for free space in the inbound buffer.

If a parity error or LLRC error is detected within a burst of this packet, then an error flag is associated with this burst, the connection is dropped (ending the packet), and the store stream will take an uncorrectable error exception after storing the data.

If a burst is received with more than the maximum 256 words allowed, then an error flag is associated with this burst, the connection is dropped (ending the packet), and the store stream will take an uncorrectable error exception after storing the data.

If a zero length packet is received, the connection is dropped and the connect_lost exception is raised. This instruction is only terminated via exceptions: connect_lost, uncorrectable_error, timeout, or force_link. In all cases, the connection is dropped when this instruction terminates. If a partial packet has been placed in the inbound buffer, a packet end is marked on an exception. **[[There is some race condition here that escapes me.]]**

The exceptions to the IN_ACCEPT opcodes are connect_lost, uncorrectable_error, and timeout.

RAISES

connect_lost, uncorrectable_error, timeout

IN_ACCEPT_

(IN_LOOPBACK *timeout*) $_{64}$C3$_{56}$ *timeout*$_{32}$00000000$_{0}$ **I**

Enter loopback mode

The HIPPI-input stream indicates to the HIPPI-output stream that it wishes to be in loopback mode. If the HIPPI-output stream acknowledges with a similar OUT_LOOPBACK command before timeout occurs, loopback mode is established. When the IOP is in loopback mode, any data and access control state present in the outbound buffer is available for transfer into the inbound buffer.

This operation is only terminated via exceptions. The connect_lost exception is raised when the connection is terminated by the out stream executing OUT_DISCONNECT. In addition, IN_LOOPBACK may be aborted with a STORE_LINK_IN operation in the store stream.

The exceptions for the IN_LOOPBACK opcode are timeout, and connect_lost.

RAISES

timeout, connect_lost

(IN_INTERCONNECT *width timeout*) $_{64}$ 67 $_{57}$ *width* $_{56}$ *timeout* $_{32}$ 00000000 $_{0}$ I

Wait for interconnect asserted

Wait for the interconnection lines for the specified HIPPI width to settle to an active state or for timeout to occur.

The exception for the IN_INTERCONNECT opcode is timeout.

RAISES

timeout

IN_INTERCONNECT_

(IN_DELAY *timeout*) $_{64}$C7$_{56}$ *timeout* $_{32}$00000000$_{0}$ I

Wait

Wait for timeout to occur. then fetch the next instruction.

There are no exceptions for the IN_DELAY opcode.

RAISES

(nothing)

Chapter 15: I/O Processor Examples

15.1 Loading Memory

The following code fragment may be used as a template for loading a segment of memory into the outbound data buffer. If an error occurs during the loading of data from the network, or the HiPPI out stream encounters an exception during the output command for this load, the driver will be notified by the link address. The notification will indicate that the `pc-offset` for the load stream is at offset 1 relative to the start of the code fragment, and the exception bit in the SSW will be set, along with the error bit that caused the exception.

<Load_Segment>≡

```
(LOAD_SEGMENT dist_en mem_t unit limit base)
(LOAD_DATA start_off end_off)
(LOAD_LINK)
```

The following code fragment loads a header of upper layer protocol in unpacked format into the outbound buffer, and then loads the actual data in an image format into the outbound buffer.

<Load_Image>≡

```
;Segment for ULP data
  (LOAD_SEGMENT ULPdist_en ULPmem_t ULPunit ULPlimit ULPbase)
  (LOAD_DATA ULPstart ULPend)
;Segment for Dataset
  (LOAD_SEGMENT dist_en mem_t unit limit base)
  (LOAD_IMAGE start_off end_off)
  (LOAD_LINK)
```

15.2 Sending Data

The following code fragment may be used as a template for making a 64 bit connection to the external HiPPI domain. When the link operation occurs, the driver must look at the exception bit in the SSW to determine whether the operation succeeded or no external device responded to the connection request.

<Make_Ring>≡

```
(OUT_RING 1 1000 Ifield)
(OUT_LINK)
```

If the `Make_Ring` code fragment fails on the `RING` instruction then the IOP must allow for a spurious connect signal. The following code fragment waits for an entire connect pulse to occur or for two milliseconds, which ever occurs first.

<Break_Ring>≡

```
(OUT_CANCEL 2000)
(OUT_DISCONNECT 100)
(OUT_LINK)
```

The following code fragment may be used to transfer a packet on the HiPPI channel. The number of words loaded by the load stream is `length`. If the load stream used `LOAD_IMAGE`, the length must be multiplied by 17/16 in order to compensate for the extra state words packed into the outbound buffer.

<Output_Packet>≡

```
(OUT_PACKET 100 length)
(OUT_LINK)
```

15.3 Receiving Data

The following code fragment returns an I-field if an external device tries to connect with the IOP before a timeout occurs. The Ifield will be stored in the pc field of the status link word.

<Listen_For_Request>≡

```
(IN_INTERCONNECT 10)
(IN_LISTEN 10000)
```

The following code fragment receives a sequence of packets from the HiPPI port. The connection is normally terminated by the sender, but can be broken by a STORE_LINK_IN operation.

<Receive_Packets>≡

```
(IN_ACCEPT 1 100)
```

15.4 Storing Memory

The following code fragment stores the upper layer protocol header in one area and the data information in an I/O buffer.

<Receive_Image>≡

```
;ULP data area
  (STORE_SEGMENT ULPdist_en ULPmem_t ULPunit ULPlimit ULPbase)
  (STORE_DATA ULPstart ULPend)
;File system IO area
  (STORE_SEGMENT dist_en mem_t unit limit base)
  (STORE_IMAGE start_off end_off)
  (STORE_LINK)
```

<iopexamples.asm>≡

```
<Load_Segment>
<Load_Image>
<Make_Ring>
<Break_Ring>
<Output_Packet>
<Listen_For_Request>
<Receive_Packets>
<Receive_Image>
```

Appendix A: Operation Encoding Summary

This chapter shows the encoding for every operation. The first column contains 64 sub columns, one for every bit in an instruction word, numbered from right to left. The second column is the assembly language prototype. Within the first column, a symbol "-" indicates that the particular bit is not used. A symbol "0" or "1" indicates a literal value encoding the operation. An alphabetic symbol shows where bits encoding an operand occur. If the operand name is "xyz", then the first letter "x" is repeated to fill the object code field. A '*' indicates an operand which is not used by the particular operation (don't care).

A.1 M OPs

```
---rrrrrsssss0000----------------------------------------------- (LOAD_FE r s)
---rrrrrsssss0001----------------------------------------------- (INT_LOADH r s)
---rrrrrsssss0010----------------------------------------------- (INT_LOADQ r s)
---rrrrrsssss0011----------------------------------------------- (INT_LOADB r s)
---rrrrrsssss0100----------------------------------------------- (LOAD r s)
---rrrrrsssss0101----------------------------------------------- (UNS_LOADH r s)
---rrrrrsssss0110----------------------------------------------- (UNS_LOADQ r s)
---00000000000110----------------------------------------------- (NOP)
---rrrrrsssss0111----------------------------------------------- (UNS_LOADB r s)
---rrrrrsssss1000----------------------------------------------- (STORE r s)
---rrrrrsssss1001----------------------------------------------- (STOREH r s)
---rrrrrsssss1010----------------------------------------------- (STOREQ r s)
---rrrrrsssss1011----------------------------------------------- (STOREB r s)
```

A.2 MC OPs

```
---rrrrrsssss1100--------------------------ddddddddddddddddddd10 (INT_LOADB_DISP r s disp)
---rrrrrsssss1100--------------------------aaaaaddddddddddddddd11 (INT_LOADB_AC_DISP r s ac disp)
---rrrrrsssss1100--------------------------dddddddddddddddddd100 (INT_LOADQ_DISP r s disp)
---rrrrrsssss1100--------------------------aaaaaddddddddddddd101 (INT_LOADQ_AC_DISP r s ac disp)
---rrrrrsssss1100--------------------------ddddddddddddddddd1000 (INT_LOADH_DISP r s disp)
---rrrrrsssss1100--------------------------aaaaadddddddddddd1001 (INT_LOADH_AC_DISP r s ac disp)
---rrrrrsssss1100--------------------------dddddddddddddddd10000 (INT_FETCH_ADD_DISP r s disp)
---rrrrrsssss1100--------------------------aaaaaddddddddddd10001 (INT_FETCH_ADD_AC_DISP r s ac disp)
---rrrrrsssss1100--------------------------dddddddddddddddd00000 (STATE_LOCK_DISP r s disp)
---rrrrrsssss1100--------------------------aaaaaddddddddddd00001 (STATE_LOCK_AC_DISP r s ac disp)
---rrrrrsssss1101--------------------------ddddddddddddddddddd10 (UNS_LOADB_DISP r s disp)
---rrrrrsssss1101--------------------------aaaaaddddddddddddddd11 (UNS_LOADB_AC_DISP r s ac disp)
---rrrrrsssss1101--------------------------dddddddddddddddddd100 (UNS_LOADQ_DISP r s disp)
---rrrrrsssss1101--------------------------aaaaaddddddddddddd101 (UNS_LOADQ_AC_DISP r s ac disp)
---rrrrrsssss1101--------------------------ddddddddddddddddd1000 (UNS_LOADH_DISP r s disp)
---rrrrrsssss1101--------------------------aaaaadddddddddddd1001 (UNS_LOADH_AC_DISP r s ac disp)
---rrrrrsssss1101--------------------------dddddddddddddddd10000 (LOAD_DISP r s disp)
---rrrrrsssss1101--------------------------aaaaaddddddddddd10001 (LOAD_AC_DISP r s ac disp)
---rrrrrsssss1101--------------------------dddddddddddddddd00000 (REG_LOAD_DISP r s disp)
---rrrrrsssss1101--------------------------aaaaaddddddddddd00001 (REG_LOAD_AC_DISP r s ac disp)
---rrrrrsssss1110--------------------------ddddddddddddddddddd10 (STOREB_DISP r s disp)
```

```
---rrrrrsssss1110---------------------------aaaaaddddddddddddddd11 (STOREB_AC_DISP r s ac disp)
---rrrrrsssss1110---------------------------ddddddddddddddddddd100 (STOREQ_DISP r s disp)
---rrrrrsssss1110---------------------------aaaaaddddddddddddddd101 (STOREQ_AC_DISP r s ac disp)
---rrrrrsssss1110---------------------------ddddddddddddddddd1000 (STOREH_DISP r s disp)
---rrrrrsssss1110---------------------------aaaaaddddddddddddd1001 (STOREH_AC_DISP r s ac disp)
---rrrrrsssss1110---------------------------ddddddddddddddddd10000 (STORE_DISP r s disp)
---rrrrrsssss1110---------------------------aaaaaddddddddddddd10001 (STORE_AC_DISP r s ac disp)
---rrrrrsssss1110---------------------------ddddddddddddddddd00000 (STATE_STORE_DISP r s disp)
---rrrrrsssss1110---------------------------00000dddddddddddd00001 (STATE_STORE_ERROR_DISP r s disp)
---rrrrrsssss1110---------------------------aaaaaddddddddddddd00001 (STATE_STORE_AC_DISP r s ac disp)
---rrrrr*****1111---------------------------xxxxxyyyyyooooo001110 (STREAM_CREATE_IMM r t u x y offset)

---**********1111---------------------------**********00000001010 (STREAM_QUIT)
---**********1111---------------------------**********00001001010 (STREAM_QUIT_PRESERVE)
---rrrrrsssss1111---------------------------ddddddddddddddddd10010 (INT_MEM_ADD_DISP r s disp)
---rrrrrsssss1111---------------------------aaaaaddddddddddddd10011 (INT_MEM_ADD_AC_DISP r s ac disp)
---*****sssss1111---------------------------**********00010000010 (DATA_MAP_FLUSH s)
---*****sssss1111---------------------------**********00011000010 (DATA_MAP_FLUSH_ANY s)
---*****sssss1111---------------------------**********00100000010 (DATA_STATE_RESTORE s)
---rrrrr*****1111---------------------------xxxxx*sssssssdd001100 (STREAM_CATCH r t x delay str)
---rrrrr*****1111---------------------------**********00ooo000100 (DATA_OPA_SAVE r opno)
---rrrrr*****1111---------------------------**********01ooo000100 (DATA_OPD_SAVE r opno)
---rrrrrsssss1111---------------------------**********10000000100 (DATA_OP_REDO r s)
---rrrrrsssss1111---------------------------*****yyyyy00000111000 (STATE_LOAD_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy00001111000 (REG_STORE_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy00010111000 (STATE_SCRUB_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy01000111000 (LOAD_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy01001111000 (REG_LOAD_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy01010111000 (UNS_LOADH_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy01100111000 (UNS_LOADQ_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy01110111000 (UNS_LOADB_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy10000111000 (INT_FETCH_ADD_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy10001111000 (STATE_LOCK_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy10010111000 (INT_LOADH_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy10011111000 (INT_MEM_ADD_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy10100111000 (INT_LOADQ_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy10110111000 (INT_LOADB_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy11000111000 (STORE_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy11001111000 (STATE_STORE_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy11010111000 (STOREH_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy11100111000 (STOREQ_INDEX r s y)
---rrrrrsssss1111---------------------------*****yyyyy11110111000 (STOREB_INDEX r s y)
---rrrrrsssss1111---------------------------lla**yyyyy00000111001 (PROBE_INDEX r s lev access y)
---rrrrrsssss1111---------------------------aaaaayyyyy00001111001 (REG_STORE_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy01000111001 (LOAD_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy01001111001 (REG_LOAD_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy01010111001 (UNS_LOADH_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy01100111001 (UNS_LOADQ_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy01110111001 (UNS_LOADB_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy10000111001 (INT_FETCH_ADD_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy10001111001 (STATE_LOCK_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy10010111001 (INT_LOADH_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy10011111001 (INT_MEM_ADD_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy10100111001 (INT_LOADQ_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy10110111001 (INT_LOADB_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy11000111001 (STORE_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------00000yyyyy11001111001 (STATE_STORE_ERROR_INDEX r s y)
---rrrrrsssss1111---------------------------aaaaayyyyy11001111001 (STATE_STORE_AC_INDEX r s ac y)
---rrrrrsssss1111---------------------------aaaaayyyyy11010111001 (STOREH_AC_INDEX r s ac y)
```

```
---rrrrrsssss1111--------------------------aaaaayyyyy11100111001 (STOREQ_AC_INDEX r s ac y)
---rrrrrsssss1111--------------------------aaaaayyyyy11110111001 (STOREB_AC_INDEX r s ac y)
---rrrrrsssss1111--------------------------dddddddddddddddd01000 (STATE_SCRUB_DISP r s disp)
---rrrrrsssss1111--------------------------dddddddddddddddd10000 (STATE_LOAD_DISP r s disp)
---rrrrrsssss1111--------------------------lla**ddddddddddd10001 (PROBE_DISP r s lev access disp)
---rrrrrsssss1111--------------------------dddddddddddddddd00000 (REG_STORE_DISP r s disp)
---rrrrrsssss1111--------------------------aaaaaddddddddddd00001 (REG_STORE_AC_DISP r s ac disp)
```

A.3 A OPs

```
-----------------*****00001**********000000--------------------- (BREAK)
-----------------*****00100*****00000000000--------------------- (STREAM_QUIT)
-----------------*****00100*****00001000000--------------------- (STREAM_QUIT_PRESERVE)
-----------------ttttt00110*****dd000000000--------------------- (STREAM_CATCH r t x delay str)
-----------------ttttt01000*****00000000000--------------------- (DOMAIN_IDENTIFIER_SAVE t)
-----------------ttttt01000*****00001000000--------------------- (STREAM_IDENTIFIER_SAVE t)
-----------------ttttt01000*****00010000000--------------------- (STREAM_CUR_SAVE t)
-----------------ttttt01000*****00011000000--------------------- (STREAM_RES_SAVE t)
-----------------ttttt01001vvvvv*****000000--------------------- (REG_MOVE t v)
-----------------ttttt01010mmmmmmmmcc000000--------------------- (LOGICAL_ONE t mask cn)
-----------------ttttt01011mmmmmmmmcc000000--------------------- (LOGICAL_ALLONE t mask cn)
-----------------ttttt011ttttttbbbbbb000000--------------------- (BIT_MASK t top bot)
-----------------ttttt11000vvvvvwwwww000000--------------------- (FLOAT_SCALB t v w)
-----------------ttttt11010vvvvvwwwww000000--------------------- (INT_SHIFT_RIGHT t v w)
-----------------ttttt11100vvvvvwwwww000000--------------------- (INT_RECIP_ERROR t v w)
-----------------ttttt11110vvvvvwwwww000000--------------------- (FLOAT_RECIP_ERROR t v w)
-----------------ttttt01010mmmmmmmmcc000001--------------------- (LOGICAL_ONE_TEST t mask cn)
-----------------ttttt01011mmmmmmmmcc000001--------------------- (LOGICAL_ALLONE_TEST t mask cn)
-----------------ttttt11010vvvvvwwwww000001--------------------- (INT_SHIFT_RIGHT_TEST t v w)
-----------------tttttvvvvvvvvvvvvvvv000010--------------------- (INT_IMM t value)
-----------------00000000000000000000000010--------------------- (NOP)
-----------------tttttuuuuuvvvvviiicc000100--------------------- (SELECT_INT t u v intselect cn)
-----------------tttttuuuuuvvvvviiicc000101--------------------- (SELECT_INT_TEST t u v intselect cn)

-----------------tttttuuuuuvvvvvfffcc000110--------------------- (SELECT_FLOAT t u v floatselect cn)

-----------------tttttuuuuuvvvvvfffcc000111--------------------- (SELECT_FLOAT_TEST t u v floatselect
cn)
-----------------tttttuuuuu00ssssssss001000--------------------- (STREAM_RESERVE t u st)
-----------------tttttuuuuu01ssssssss001000--------------------- (STREAM_RESERVE_UPTO t u st)
-----------------tttttuuuuu1000ssssss001000--------------------- (SHIFT_LEFT_IMM t u sh)
-----------------tttttuuuuu1100000000001000--------------------- (FLOAT_NEAR t u)
-----------------tttttuuuuu1100000001001000--------------------- (FLOAT_CHOP t u)
-----------------tttttuuuuu1100000010001000--------------------- (FLOAT_FLOOR t u)
-----------------tttttuuuuu1100000011001000--------------------- (FLOAT_CEIL t u)
-----------------tttttuuuuu1100000100001000--------------------- (INT_NEAR t u)
-----------------tttttuuuuu1100000101001000--------------------- (INT_CHOP t u)
-----------------tttttuuuuu1100000110001000--------------------- (INT_FLOOR t u)
-----------------tttttuuuuu1100000111001000--------------------- (INT_CEIL t u)
-----------------tttttuuuuu1100001000001000--------------------- (UNS_NEAR t u)
-----------------tttttuuuuu1100001001001000--------------------- (UNS_CHOP t u)
-----------------tttttuuuuu1100001010001000--------------------- (UNS_FLOOR t u)
-----------------tttttuuuuu1100001011001000--------------------- (UNS_CEIL t u)
-----------------tttttuuuuu1100001100001000--------------------- (FLOAT_ROUND t u)
-----------------tttttuuuuu1100001101001000--------------------- (INT_ROUND t u)
-----------------tttttuuuuu1100001110001000--------------------- (UNS_ROUND t u)
```

```
----------------ttttttuuuuu1100011100001000-------------------- (FLOAT_REAL t u)
----------------tttttuuuuu1100011110001000-------------------- (FLOAT_INT t u)
----------------tttttuuuuu1100011111001000-------------------- (FLOAT_UNS t u)
----------------tttttuuuuu11010aaaaa001000-------------------- (PTR_SET_AC t u ac)
----------------tttttuuuuu1101100000001000-------------------- (BIT_MAT_TRANSPOSE t u)
----------------ttttt*****1101110000001000-------------------- (COUNT_ISSUES t)
----------------ttttt*****1101110001001000-------------------- (COUNT_MEMREFS t)
----------------ttttt*****1101110010001000-------------------- (COUNT_STREAMS t)
----------------ttttt*****1101110011001000-------------------- (COUNT_CONCURRENCY t)
----------------ttttt*****11011101ee001000-------------------- (COUNT_EVENTS t ec)
----------------ttttt*****1101111000001000-------------------- (COUNT_PHANTOMS t)
----------------ttttt*****1101111001001000-------------------- (COUNT_READY t)
----------------ttttt*****1101111100001000-------------------- (COUNT_SELECT_SAVE t)
----------------tttttuuuuu00ssssssss001001-------------------- (STREAM_RESERVE_TEST t u st)
----------------tttttuuuuu01ssssssss001001-------------------- (STREAM_RESERVE_UPTO_TEST t u st)
----------------tttttuuuuu1000ssssss001001-------------------- (SHIFT_LEFT_IMM_TEST t u sh)
----------------tttttuuuuu1100000100001001-------------------- (INT_NEAR_TEST t u)
----------------tttttuuuuu1100000101001001-------------------- (INT_CHOP_TEST t u)
----------------tttttuuuuu1100000110001001-------------------- (INT_FLOOR_TEST t u)
----------------tttttuuuuu1100000111001001-------------------- (INT_CEIL_TEST t u)
----------------tttttuuuuu1100001000001001-------------------- (UNS_NEAR_TEST t u)
----------------tttttuuuuu1100001001001001-------------------- (UNS_CHOP_TEST t u)
----------------tttttuuuuu1100001010001001-------------------- (UNS_FLOOR_TEST t u)
----------------tttttuuuuu1100001011001001-------------------- (UNS_CEIL_TEST t u)
----------------tttttuuuuu1100001101001001-------------------- (INT_ROUND_TEST t u)
----------------tttttuuuuu1100001110001001-------------------- (UNS_ROUND_TEST t u)
----------------***00uuuuu*****00000001010-------------------- (PROGRAM_STATE_RESTORE u)
----------------***00uuuuu*****00010001010-------------------- (PROGRAM_MAP_FLUSH u)
----------------***00uuuuu*****00011001010-------------------- (PROGRAM_MAP_FLUSH_ANY u)
----------------***00uuuuu*****00100001010-------------------- (PROGRAM_CACHE_FLUSH u)
----------------***00uuuuu*****00101001010-------------------- (PROGRAM_CACHE_FLUSH_L1 u)
----------------***00uuuuu*****00111001010-------------------- (PROGRAM_CACHE_FLUSH_ANY u)
----------------***00uuuuu*****01001001010-------------------- (EXCEPTION_RESTORE u)
----------------***00uuuuu*****01010001010-------------------- (SSW_RESTORE u)
----------------***00uuuuu*****01100001010-------------------- (DOMAIN_LEAVE u)
----------------***00**********01101001010-------------------- (DOMAIN_ENTER)
----------------***00uuuuu*****10000001010-------------------- (COUNT_SELECT_RESTORE u)
----------------ttt01uuuuu**********001010-------------------- (TARGET_RESTORE tn u)
----------------ttt10ooooooooooooooo001010-------------------- (TARGET_DISP tn offset)
----------------ttt11uuuuu0000000000001010-------------------- (TARGET_INDEX tn u)
----------------tttttuuuuuoooooooooo001011-------------------- (STREAM_CREATE_IMM r t u x y offset)

----------------tttttuuuuuvvvvv00000001100-------------------- (BIT_NIMP t u v)
----------------tttttuuuuuvvvvv00001001100-------------------- (BIT_AND t u v)
----------------tttttuuuuuvvvvv00010001100-------------------- (BIT_XOR t u v)
----------------tttttuuuuuvvvvv00011001100-------------------- (BIT_OR t u v)
----------------tttttuuuuuvvvvv00100001100-------------------- (BIT_NOR t u v)
----------------tttttuuuuuvvvvv00101001100-------------------- (BIT_XNOR t u v)
----------------tttttuuuuuvvvvv00110001100-------------------- (BIT_NAND t u v)
----------------tttttuuuuuvvvvv00111001100-------------------- (BIT_IMP t u v)
----------------tttttuuuuuvvvvv01000001100-------------------- (BIT_ODD_NIMP t u v)
----------------tttttuuuuuvvvvv01001001100-------------------- (BIT_ODD_AND t u v)
----------------tttttuuuuuvvvvv01010001100-------------------- (BIT_ODD_XOR t u v)
----------------tttttuuuuuvvvvv01011001100-------------------- (BIT_ODD_OR t u v)
----------------tttttuuuuu0000001111001100-------------------- (BIT_TALLY t u)
----------------tttttuuuuuvvvvv10000001100-------------------- (BIT_MAT_OR t u v)
----------------tttttuuuuuvvvvv10001001100-------------------- (BIT_MAT_XOR t u v)
----------------tttttuuuuuvvvvv10100001100-------------------- (BIT_PACK t u v)
----------------tttttuuuuuvvvvv10101001100-------------------- (BIT_UNPACK_1 t u v)
```

```
----------------tttttuuuuuvvvvv10110001100---------------------- (BIT_UNPACK_2 t u v)
----------------tttttuuuuuvvvvv10111001100---------------------- (BIT_UNPACK_3 t u v)
----------------tttttuuuuuvvvvv00000001101---------------------- (BIT_NIMP_TEST t u v)
----------------tttttuuuuuvvvvv00001001101---------------------- (BIT_AND_TEST t u v)
----------------tttttuuuuuvvvvv00010001101---------------------- (BIT_XOR_TEST t u v)
----------------tttttuuuuuvvvvv00011001101---------------------- (BIT_OR_TEST t u v)
----------------tttttuuuuuvvvvv00100001101---------------------- (BIT_NOR_TEST t u v)
----------------tttttuuuuuvvvvv00101001101---------------------- (BIT_XNOR_TEST t u v)
----------------tttttuuuuuvvvvv00110001101---------------------- (BIT_NAND_TEST t u v)
----------------tttttuuuuuvvvvv00111001101---------------------- (BIT_IMP_TEST t u v)
----------------tttttuuuuuvvvvv01000001101---------------------- (BIT_ODD_NIMP_TEST t u v)
----------------tttttuuuuuvvvvv01001001101---------------------- (BIT_ODD_AND_TEST t u v)
----------------tttttuuuuuvvvvv01010001101---------------------- (BIT_ODD_XOR_TEST t u v)
----------------tttttuuuuuvvvvv01011001101---------------------- (BIT_ODD_OR_TEST t u v)
----------------tttttuuuuu0000001111001101---------------------- (BIT_TALLY_TEST t u)
----------------tttttuuuuuvvvvv10000001110---------------------- (FLOAT_ADD t u v)
----------------tttttuuuuuvvvvv10001001110---------------------- (FLOAT_SUB t u v)
----------------tttttuuuuuvvvvv10010001110---------------------- (FLOAT_MIN t u v)
----------------tttttuuuuuvvvvv10011001110---------------------- (FLOAT_MAX t u v)
----------------tttttuuuuuvvvvv10100001110---------------------- (FLOAT_MMIN t u v)
----------------tttttuuuuuvvvvv10101001110---------------------- (FLOAT_MMAX t u v)
----------------ttttt*****vvvvv11001001110---------------------- (REAL_FLOAT t v)
----------------tttttuuuuuvvvvv11100001110---------------------- (INT_ADD t u v)
----------------tttttuuuuuvvvvv11101001110---------------------- (INT_SUB t u v)
----------------tttttuuuuuvvvvv11110001110---------------------- (INT_MIN t u v)
----------------tttttuuuuuvvvvv11111001110---------------------- (INT_MAX t u v)
----------------tttttuuuuuvvvvv10001001111---------------------- (FLOAT_CMP_TEST t u v)
----------------tttttuuuuuvvvvv10010001111---------------------- (FLOAT_MIN_TEST t u v)
----------------tttttuuuuuvvvvv10011001111---------------------- (FLOAT_MAX_TEST t u v)
----------------tttttuuuuuvvvvv10100001111---------------------- (FLOAT_MMIN_TEST t u v)
----------------tttttuuuuuvvvvv10101001111---------------------- (FLOAT_MMAX_TEST t u v)
----------------tttttuuuuuvvvvv11100001111---------------------- (INT_ADD_TEST t u v)
----------------tttttuuuuuvvvvv11101001111---------------------- (INT_SUB_TEST t u v)
----------------tttttuuuuuvvvvv11110001111---------------------- (INT_MIN_TEST t u v)
----------------tttttuuuuuvvvvv11111001111---------------------- (INT_MAX_TEST t u v)
----------------tttttuuuuuvvvvvwwwww010100---------------------- (SHIFT_PAIR_LEFT t u v w)
----------------tttttuuuuuvvvvvwwwww010101---------------------- (SHIFT_PAIR_LEFT_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww010110---------------------- (SHIFT_PAIR_RIGHT t u v w)
----------------tttttuuuuuvvvvvwwwww010111---------------------- (SHIFT_PAIR_RIGHT_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww011000---------------------- (INT_DIV_CHOP t u v w)
----------------tttttuuuuuvvvvvwwwww011001---------------------- (INT_DIV_CHOP_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww011010---------------------- (INT_DIV_FLOOR t u v w)
----------------tttttuuuuuvvvvvwwwww011011---------------------- (INT_DIV_FLOOR_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww011100---------------------- (UNS_DIV t u v w)
----------------tttttuuuuuvvvvvwwwww011101---------------------- (UNS_DIV_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww011110---------------------- (FLOAT_DIV t u v w)
----------------tttttuuuuuvvvvvwwwww011111---------------------- (FLOAT_SQRT t u v w)
----------------tttttuuuuu0vvvvvvvvv100000---------------------- (INT_ADD_IMM t u value)
----------------tttttuuuuu1vvvvvvvvv100000---------------------- (INT_SUB_IMM t u value)
----------------tttttuuuuu0vvvvvvvvv100001---------------------- (INT_ADD_IMM_TEST t u value)
----------------tttttuuuuu1vvvvvvvvv100001---------------------- (INT_SUB_IMM_TEST t u value)
----------------tttttuuuuuvvvvvwwwww100110---------------------- (BIT_MERGE t u v w)
----------------tttttuuuuuvvvvvwwwww100111---------------------- (BIT_MERGE_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww101000---------------------- (INT_ADD_MUL t u v w)
----------------tttttuuuuuvvvvvwwwww101001---------------------- (INT_ADD_MUL_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww101010---------------------- (INT_SUB_MUL t u v w)
----------------tttttuuuuuvvvvvwwwww101011---------------------- (INT_SUB_MUL_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww101100---------------------- (UNS_ADD_MUL_UPPER t u v w)
----------------tttttuuuuuvvvvvwwwww101101---------------------- (UNS_ADD_MUL_UPPER_TEST t u v w)
```

A.3 A OPs

```
----------------tttttuuuuuvvvvvwwwww101110---------------------- (INT_SUB_MUL_REV t u v w)
----------------tttttuuuuuvvvvvwwwww101111---------------------- (INT_SUB_MUL_REV_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww110000---------------------- (FLOAT_ADD_MUL t u v w)
----------------tttttuuuuuvvvvvwwwww110010---------------------- (FLOAT_SUB_MUL t u v w)
----------------tttttuuuuuvvvvvwwwww110100---------------------- (FLOAT_MUL_LOWER t u v w)
----------------tttttuuuuuvvvvvwwwww110110---------------------- (FLOAT_SUB_MUL_REV t u v w)
----------------tttttuuuuuvvvvvwwwww111000---------------------- (FLOAT_ITER t u v w)
----------------tttttuuuuuvvvvvwwwww111010---------------------- (FLOAT_DIV_APPROX t u v w)
----------------tttttuuuuuvvvvvwwwww111011---------------------- (FLOAT_SQRT_APPROX_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww111100---------------------- (FLOAT_DIV_ERROR t u v w)
----------------tttttuuuuuvvvvvwwwww111101---------------------- (FLOAT_SQRT_ERROR_TEST t u v w)
----------------tttttuuuuuvvvvvwwwww111111---------------------- (FLOAT_RSQRT_ERROR_TEST t u v w)
```

A.4 C OPs

```
-------------------------------------------*****yyyyy00001000000 (FLOAT_APPROX_RESTORE y)
-------------------------------------------xxxxxyyyyy00010000000 (REG_MOVE x y)
-------------------------------------------xxxxxttttt00011000000 (TRAP_SAVE x tr)
-------------------------------------------xxxxxyyyyy00100000000 (BIT_LEFT_ONES x y)
-------------------------------------------xxxxxyyyyy00101000000 (BIT_LEFT_ZEROS x y)
-------------------------------------------xxxxxyyyyy00110000000 (BIT_RIGHT_ONES x y)
-------------------------------------------xxxxxyyyyy00111000000 (BIT_RIGHT_ZEROS x y)
-------------------------------------------xxxxxyyyyy01000000000 (INT_RECIP_APPROX x y)
-------------------------------------------xxxxxyyyyy01001000000 (INT_RSQRT_APPROX x y)
-------------------------------------------xxxxxyyyyy01010000000 (STREAM_COUNT_INST_RESTORE x y)
-------------------------------------------xxxxxyyyyy01011000000 (INT_LOGB x y)
-------------------------------------------xxxxxyyyyy01100000000 (FLOAT_RECIP_APPROX x y)
-------------------------------------------xxxxxyyyyy01101000000 (FLOAT_RSQRT_APPROX x y)
-------------------------------------------xxxxxyyyyy01110000000 (INT_RECIP_SHIFT x y)
-------------------------------------------xxxxxyyyyy01111000000 (UNS_RECIP_SHIFT x y)
-------------------------------------------xxxxx*****11000000000 (STREAM_COUNT_INST x)
-------------------------------------------000000000011001000000 (NOP)
-------------------------------------------xxxxxyyyyy11001000000 (CLOCK x y)
-------------------------------------------xxxxx*****11100000000 (EXCEPTION_SAVE x)
-------------------------------------------xxxxx*****11101000000 (RESULTCODE_SAVE x)
-------------------------------------------xxxxx*****11111000000 (STREAM_LOOKAHEAD_SAVE x)
-------------------------------------------xxxxxyyyyy00100000001 (BIT_LEFT_ONES_TEST x y)
-------------------------------------------xxxxxyyyyy00101000001 (BIT_LEFT_ZEROS_TEST x y)
-------------------------------------------xxxxxyyyyy00110000001 (BIT_RIGHT_ONES_TEST x y)
-------------------------------------------xxxxxyyyyy00111000001 (BIT_RIGHT_ZEROS_TEST x y)
-------------------------------------------xxxxxyyyyy01011000001 (INT_LOGB_TEST x y)
-------------------------------------------xxxxxyyyyy01100000001 (FLOAT_RECIP_APPROX_TEST x y)
-------------------------------------------xxxxxyyyyy01101000001 (FLOAT_RSQRT_APPROX_TEST x y)
-------------------------------------------xxxxxyyyyy01110000001 (INT_RECIP_SHIFT_TEST x y)
-------------------------------------------xxxxxyyyyy01111000001 (UNS_RECIP_SHIFT_TEST x y)
-------------------------------------------xxxxxyyyyyzzzzz000010 (ROTATE_RIGHT x y z)
-------------------------------------------xxxxxyyyyyzzzzz000011 (ROTATE_RIGHT_TEST x y z)
-------------------------------------------xxxxxyyyyyvvvvv000100 (INT_ADD_IMM x y value)
-------------------------------------------xxxxxyyyyyvvvvv000101 (INT_ADD_IMM_TEST x y value)
-------------------------------------------xxxxxyyyyyvvvvv000110 (INT_SUB_IMM x y value)
-------------------------------------------xxxxxyyyyyvvvvv000111 (INT_SUB_IMM_TEST x y value)
-------------------------------------------xxxxxyyyyyzzzzz001100 (FLOAT_ADD x y z)
-------------------------------------------xxxxxyyyyyzzzzz001110 (FLOAT_SUB x y z)
-------------------------------------------xxxxxyyyyyzzzzz001111 (FLOAT_CMP_TEST x y z)
-------------------------------------------xxxxxyyyyyzzzzz010000 (BIT_NIMP x y z)
-------------------------------------------xxxxxyyyyyzzzzz010001 (BIT_NIMP_TEST x y z)
```

```
-------------------------------------------xxxxxyyyyyzzzzz010010 (BIT_AND x y z)
-------------------------------------------xxxxxyyyyyzzzzz010011 (BIT_AND_TEST x y z)
-------------------------------------------xxxxxyyyyyzzzzz010100 (BIT_XOR x y z)
-------------------------------------------xxxxxyyyyyzzzzz010101 (BIT_XOR_TEST x y z)
-------------------------------------------xxxxxyyyyyzzzzz010110 (BIT_OR x y z)
-------------------------------------------xxxxxyyyyyzzzzz010111 (BIT_OR_TEST x y z)
-------------------------------------------xxxxxyyyyyzzzzz011000 (SHIFT_LEFT x y z)
-------------------------------------------xxxxxyyyyyzzzzz011001 (SHIFT_LEFT_TEST x y z)
-------------------------------------------xxxxxyyyyyzzzzz011010 (ROTATE_LEFT x y z)
-------------------------------------------xxxxxyyyyyzzzzz011011 (ROTATE_LEFT_TEST x y z)
-------------------------------------------xxxxxyyyyyzzzzz011100 (UNS_SHIFT_RIGHT x y z)
-------------------------------------------xxxxxyyyyyzzzzz011101 (UNS_SHIFT_RIGHT_TEST x y z)
-------------------------------------------xxxxxyyyyyzzzzz011110 (INT_SHIFT_RIGHT x y z)
-------------------------------------------xxxxxyyyyyzzzzz011111 (INT_SHIFT_RIGHT_TEST x y z)
-------------------------------------------xxxxxyyyyyaaaaa100000 (PTR_SET_AC x y ac)
-------------------------------------------xxxxxyyyyyzzzzz100001 (UNS_ADD_CARRY_TEST x y z)
-------------------------------------------oooooyyyyyttttt100010 (TRAP_RESTORE tr y)
-------------------------------------------xxxxxyyyyyzzzzz100011 (UNS_SUB_CARRY_TEST x y z)
-------------------------------------------xxxxxyyyyyzzzzz100100 (INT_ADD x y z)
-------------------------------------------xxxxxyyyyyzzzzz100101 (INT_ADD_TEST x y z)
-------------------------------------------xxxxxyyyyyzzzzz100110 (INT_SUB x y z)
-------------------------------------------xxxxxyyyyyzzzzz100111 (INT_SUB_TEST x y z)
-------------------------------------------xxxxx000oooooo0110ooo (SSH_DISP x offset)
-------------------------------------------mmmmmmmmccoooo1110ooo (SKIP mask cn offset)
-------------------------------------------xxxxx000oo11110110ttt (TARGET_SAVE x tn)
-------------------------------------------mmmmmmmmcc11111110ttt (JUMP mask cn tn)
-------------------------------------------ooooo0011100000110000 (LEVEL_ENTER lev)
-------------------------------------------ooooo0011111110110ttt (LEVEL_RTN lev tn)
-------------------------------------------mmmmmmmmccoooo0111ooo (SKIP_OFTEN mask cn offset)
-------------------------------------------mmmmmmmmccoooo1111ooo (SKIP_SELDOM mask cn offset)
-------------------------------------------mmmmmmmmcc11110111ttt (JUMP_OFTEN mask cn tn)
-------------------------------------------mmmmmmmmcc11111111ttt (JUMP_SELDOM mask cn tn)
```

A.5 MAC OPs

```
---00000000000000010000ccccccccccccccccccccccccccccccccccccccccc (DEBUG cookie)
---rrrrr00000000011111000000000000000000000000000000000000000000 (DEBUG_REG r)
---oooooooooo1111ooooo00100ooooo0000000000oooooooooo0000001010 (STREAM_QUIT)
---oooooooooo1111ooooo00100ooooo0000100000oooooooooo0001001010 (STREAM_QUIT_PRESERVE)
---rrrrrooooo1111ttttt00110oooooodd000000000xxxxxosssssssdd001100 (STREAM_CATCH r t x delay str)
---rrrrrooooo1111tttttuuuuuoooooooooo001011xxxxxyyyyyooooo001110 (STREAM_CREATE_IMM r t u x y offset)
```

A.6 I OPs

```
000000000dmm0000uuuuuuuu0111111111111111100000bbbbbbbbbbbbbbbbbb (INST_SEGMENT dist_en m_type unit limit
base)
0001000000000000000000000000000000000000000000000000000000000000 (LOAD_LINK)
000100010dmm0000uuuuuuuu0111111111111111100000bbbbbbbbbbbbbbbbbb (LOAD_SEGMENT dist_en m_type unit limit
base)
0001001000000000000000000000000000000000000000000000000000000000 (LOAD_ERR_OFFSET)
0001001100000000000000000000000000000000000000000000000000000000 (LOAD_FLUSH)
0001010000000000000000000000000000000000000000000000000000000000 (LOAD_LINK_OUT)
0001010100000000000000000000000000000000000000000000000000000000 (LOAD_END_PACKET)
```

```
0010sssssssssssssssssssssssssssssseeeeeeeeeeeeeeeeeeeeeeeeeeeeee (LOAD_DATA start_offset end_offset)
0011sssssssssssssssssssssssssssssseeeeeeeeeeeeeeeeeeeeeeeeeeeeee (LOAD_IMAGE start_offset end_offset)

0100000100000000000000000000000000000000000000000000000000000000 (OUT_LINK)
01000100tttttttttttttttttttttttt00000000000000000000000000000000 (OUT_DISCONNECT timeout)
01000101tttttttttttttttttttttttt00000000000000000000000000000000 (OUT_CANCEL timeout)
01000111tttttttttttttttttttttttt00000000000000000000000000000000 (OUT_DELAY timeout)
01000111tttttttttttttttttttttttt00000000000000000000000000000001 (OUT_RESET timeout)
0100100000000000000000000000000000000000000000000000000000000000 (OUT_LINK_LOAD)
01001001ttttttttttttttttttttttttssssssssssssssssssssssssssssssss (OUT_PACKET timeout size)
0100101000000000000000000000000010000000000000000000000000000001 (OUT_LOOPMODE)
01001010tttttttttttttttttttttttt00000000000000000000000000000000 (OUT_LOOPBACK timeout)
0100111wtttttttttttttttttttttttt00000000000000000000000000000000 (OUT_INTERCONNECT width timeout)
010s001wttttttttttttttttttttttttIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII (OUT_RING swap width timeout Ifield)

1000000000000000000000000000000000000000000000000000000000000000 (STORE_LINK)
100000010dmm0000uuuuuuuu0111111111111111100000bbbbbbbbbbbbbbbbbb (STORE_SEGMENT dist_en m_type unit limit
base)
1000001000000000000000000000000000000000000000000000000000000000 (STORE_FLUSH)
1000001100000000000000000000000000000000000000000000000000000000 (STORE_ERR_OFFSET)
1000010000000000000000000000000000000000000000000000000000000000 (STORE_LINK_IN)
1000010100000000000000000000000000000000000000000000000000000000 (STORE_END_PACKET)
1000011000000000000000000000000000000000000000000000000000000000 (STORE_END_SEGMENT)
1001sssssssssssssssssssssssssssssseeeeeeeeeeeeeeeeeeeeeeeeeeeeee (STORE_REPLICATE start_offset end_offset)

1010sssssssssssssssssssssssssssssseeeeeeeeeeeeeeeeeeeeeeeeeeeeee (STORE_DATA start_offset end_offset)

1011sssssssssssssssssssssssssssssseeeeeeeeeeeeeeeeeeeeeeeeeeeeee (STORE_IMAGE start_offset end_offset)

1100000100000000000000000000000000000000000000000000000000000000 (IN_LINK)
11000011tttttttttttttttttttttttt00000000000000000000000000000000 (IN_LOOPBACK timeout)
1100010000000000000000000000000000000000000000000000000000000000 (IN_REJECT)
11000101tttttttttttttttttttttttt00000000000000000000000000000000 (IN_LISTEN timeout)
11000111tttttttttttttttttttttttt00000000000000000000000000000000 (IN_DELAY timeout)
1100101100000000000000000000000000000000000000000000000000000000 (IN_LINK_STORE)
1100111wtttttttttttttttttttttttt00000000000000000000000000000000 (IN_INTERCONNECT width timeout)
110s100wtttttttttttttttttttttttt00000000000000000000000000000000 (IN_ACCEPT swap width timeout)
```

Appendix B: Processor State

The following table describes all of the state information maintained by a processor. The rows describe what state information is maintained and whether user, supervisor, and IPL privilege can directly read (abbreviated "r") or write (abbreviated "w"), that state. The asterisk ("*") indicates that the state cannot be written, but can be indirectly modified. Kernel level has the same capabilities as supervisor level. An unfilled entry is the same as the one above it.

per	LEV_USER	LEV_SUPER	LEV_IPL	number	bits	what	reference
stream	rw	rw	rw	1	64	stream status word	§2.1
				1	64	exception register	§9.1
				1	64	result code register	§9.1
				31	64	general purpose registers	§1.3
				8	32	target registers	§2.2
				1	16	instruction count register	§10
	-	rw	rw	1	4	protection domain	§8.2
	-	-	-	1	2	stream level	§8.1
pd	r*	r*	r*	1	56	instruction issue counter	§10
				1	56	memory reference counter	§10
				1	56	stream counter	§10
				1	56	concurrency counter	§10
				4	64	selectable event counters	§10.2
	-	rw	rw	1	64	data state descriptor	§6.2
				1	64	program state descriptor	§7.1
				16,384	64	data address map entries	§6.2
				8,192	64	program address map entries	§7
	r*	r*	r*	1	7	stream reserved, SRES$_D$	§2
				1	7	stream current, SCUR$_D$	§2
proc	rw	rw	rw	384	64	trap registers	§9.2
				512	64	data control registers	§6.3
				512	64	data value registers	§6.3
	r	r*	r*	132	32	program address TLB entries	§7
				1024	64	data address TLB entries	§6.2
	r	r	rw	256	32	reciprocal table	
				256	32	reciprocal square root table	
	r	r	r	1	56	phantom counter	§10.2
				1	56	ready counter	§10.2
				1	64	clock	§10

B.0 Storing Memory

Appendix C: GF(2) Addressing Matrices

C.1 Scrambling Matrices

This is the GF(2) matrix used for address scrambling:

```
1 0 1 1 0 0 0 0 1 1 0 0 0 1 1 0 1 0 1
1 0 1 0 0 1 0 0 0 1 1 1 1 0 1 0 1 0 1
0 0 1 1 0 0 1 1 1 0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 1 1 1 0 0 0 1 0 1 1 0 0 1 0
0 0 0 1 1 1 0 1 0 0 0 0 1 0 0 1 0 1 0
0 1 0 1 0 0 0 1 1 0 0 0 1 0 1 0 0 0 1
0 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 1 0
0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 1 1 1 0
0 1 1 0 0 1 1 1 1 0 1 1 0 0 0 1 0 1 0
0 0 1 1 1 1 1 1 1 0 0 0 1 1 1 1 0 1 0
1 0 0 0 1 1 0 1 0 0 0 0 0 1 0 1 1 0 1
0 1 1 0 1 1 0 1 1 1 1 1 0 1 1 1 0 0 1
0 0 1 1 0 1 1 0 0 0 0 1 1 1 0 1 1 1 0
0 0 0 1 1 1 1 0 1 0 1 1 0 1 1 0 0 1 0
0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 0 1
0 0 0 0 0 1 1 0 0 1 1 1 1 1 0 0 1 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0
0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 1 1 1 0
0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1
0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

This is the GF(2) inverse matrix:

```
1 0 0 0 1 1 1 1 0 1 0 1 0 1 0 1 0 1 1
0 1 1 1 0 1 1 1 0 0 1 0 0 0 0 1 0 1 0
0 0 1 1 1 0 0 0 1 0 1 1 0 0 1 1 1 0 1
0 0 0 1 1 1 0 0 1 1 1 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 1 1
0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 1
0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 0 1 0 0
0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 0
0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 1 0 1 1
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

This is the data single-bit error syndrome table. If a syndrome is found in this table, then the bit 4 * *row* + *col* is in error. If the syndrome is zero, there is no error. Otherwise, there is an uncorrectable error. Bits 71–64 are the ecc bits.

```
0x80 0x40 0x20 0x10
0x08 0x04 0x02 0x01
0xc6 0xe1 0xe2 0xd1
0xc9 0xd2 0xe4 0xe8
0xd4 0xca 0xc5 0xc3
0x86 0xa1 0xa2 0x91
0x89 0x92 0xa4 0xa8
0x94 0x8a 0x85 0x83
0x8c 0xb0 0xa0 0x90
0x88 0x84 0x82 0x81
0x46 0x61 0x62 0x51
0x49 0x52 0x64 0x68
0x54 0x4a 0x45 0x43
0x4c 0x70 0x60 0x50
0x48 0x44 0x42 0x41
0x06 0x21 0x22 0x11
0x09 0x12 0x24 0x28
0x14 0x0a 0x05 0x03
```

This is the access state single-bit error syndrome table. If a syndrome is found in this table, then the bit *row* is in error. If the syndrome is zero, there is no error. Otherwise, there is an uncorrectable error. Bits 7–4 are the ecc bits. Bits 3–0 are the access state.

C.1 Scrambling Matrices

0x8
0x4
0x2
0x1
0x7
0xb
0xd
0xe

Index